Hugues Bersini

La programmation orientée objet

Cours et exercices en UML 2 avec Java 5, C# 2, C++, Python, PHP 5 et LINQ

EYROLLES

La programmation orientée objet

DANS LA MÊME COLLECTION

C. DELANNOY. – **Programmer en Java.** *Java 5 et 6.*
N°12232, 5e édition, 2007, 800 pages avec CD-Rom.

J.-B. BOICHAT. – **Apprendre Java et C++ en parallèle**.
N° 12403, 4e édition, 2008, 600 pages avec CD-Rom.

A. TASSO. – **Le livre de Java premier langage.**
Avec 80 exercices corrigés.
N°12376, 5e édition, 2008, 520 pages avec CD-Rom.

C. DABANCOURT. – **Apprendre à programmer.**
Algorithmes et conception objet - BTS, Deug, IUT, licence
N°12350, 2e édition, 2008, 296.

P. ROQUES. – **UML 2 par la pratique.** *Étude de cas et exercices corrigés.*
N°12322, 6e édition, 2008, 368.

A. TASSO. – **Apprendre à programmer en ActionScript 3.**
N°12199, 2008, 438 pages.

A. BRILLANT. – **XML**. *Cours et exercices*.
N°12151, 2007, 282 pages.

C. DELANNOY. – **C++ pour les programmeurs C**.
N°12231, 6e édition, 2007, 602 pages.

C. SOUTOU. – **UML 2 pour les bases de données.**
Avec 20 exercices corrigés.
N°12091, 2007, 314 pages.

X BLANC, I. MOUNIER. – **UML 2 pour les développeurs.**
N°12029, 2006, 202 pages

H. SUTTER (trad. T. PETILLON). – **Mieux programmer en C++**
N°09224, 2001, 215 pages.

P. HAGGAR (trad. T. THAUREAUX). –
Mieux programmer en Java
N°09171, 2000, 225 pages.

CHEZ LE MÊME ÉDITEUR

B. MEYER. – **Conception et programmation orientées objet.**
N°12270, 2008, 1222 pages (Collection Blanche).

T. ZIADÉ. – **Programmation Python**.
N°11677, 2006, 530 pages (Collection Blanche).

P. ROQUES. – **UML 2.** *Modéliser une application web.*
N°11770, 2006, 236 pages (coll. *Cahiers du programmeur*).

P. ROQUES, F. VALLÉE. – **UML 2 en action**.
De l'analyse des besoins à la conception.
N°12104, 4e édition 2007, 382 pages.

E. PUYBARET. – **Swing.**
N°12019, 2007, 500 pages (coll. *Cahiers du programmeur*)

E. PUYBARET. – **Java 1.4 et 5.0.**
N°11916, 3e édition 2006, 400 pages
(coll. *Cahiers du programmeur*)

S POWERS. – **Débuter en JavaScript**
N°12093, 2007, 386 pages

T. TEMPLIER, A. GOUGEON. – **JavaScript pour le Web 2.0**
N°12009, 2007, 492 pages

J. ZELDMAN. – **Design web : utiliser les standards**, CSS et XHTML.
N°12026, 2e édition 2006, 444 pages.

X. BRIFFAULT, S. DUCASSE. – **Programmation Squeak**
N°11023, 2001, 328 pages.

J.-L. BÉNARD, L. BOSSAVIT , R.MÉDINA , D. WILLIAMS.
– **L'Extreme Programming, avec deux études de cas.**
N°11051, 2002, 300 pages.

P. RIGAUX, A. ROCHFELD. – **Traité de modélisation objet**.
N°11035, 2002, 308 pages.

Hugues Bersini

La programmation orientée objet

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Cet ouvrage est la quatrième édition avec mise à jour et changement de titre de l'ouvrage de Hugues Bersini et Ivan Wellesz paru à l'origine sous le titre « L'Orienté objet » *(ISBN 978-2-212-12084-8)*

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2009, ISBN : 978-2-212-12441-5

Table des matières

Avant-propos 1

L'orientation objet en deux mots 2
Objectifs de l'ouvrage 5
Plan de l'ouvrage 6
À qui s'adresse ce livre ? 7

CHAPITRE 1
Principes de base : quel objet pour l'informatique ? 9

Le trio <entité, attribut, valeur> 10
Stockage des objets en mémoire 11
L'objet dans sa version passive 15
L'objet dans sa version active 17
Introduction à la notion de classe 19
Des objets en interaction 21
Des objets soumis à une hiérarchie 24
Polymorphisme 26
Héritage bien reçu 27
Exercices 27

CHAPITRE 2
Un objet sans classe… n'a pas de classe 29

Constitution d'une classe d'objets 30
La classe comme module fonctionnel 33
La classe comme garante de son bon usage 36
La classe comme module opérationnel 37

Un premier petit programme complet dans les cinq langages 39
La classe et la logistique de développement 50
Exercices 52

CHAPITRE 3

Du faire savoir au savoir-faire… du procédural à l'OO 57

Objectif objet : les aventures de l'OO 58
Mise en pratique 60
Analyse 60
Conception 62
Impacts de l'orientation objet 62

CHAPITRE 4

Ici Londres : les objets parlent aux objets 65

Envois de messages 66
Association de classes 67
Dépendance de classes 68
Réaction en chaîne de messages 70
Exercices 71

CHAPITRE 5

Collaboration entre classes 73

Pour en finir avec la lutte des classes 74
La compilation Java : effet domino 76
En C#, en Python, PHP 5 et en C++ 77
De l'association unidirectionnelle à l'association bidirectionnelle 79
Auto-association 82
Package et namespace 83
Exercices 86

CHAPITRE 6

Méthodes ou messages ? 87

Passage d'arguments prédéfinis dans les messages 88
Passage d'argument objet dans les messages 95
Une méthode est-elle d'office un message ? 102
La mondialisation des messages 104
Exercices 105

CHAPITRE 7
L’encapsulation des attributs 109

Accès aux attributs d’un objet 110
Encapsulation : pourquoi faire ? 115
Exercices 120

CHAPITRE 8
Les classes et leur jardin secret 123

Encapsulation des méthodes 124
Les niveaux intermédiaires d’encapsulation 127
Afin d’éviter l’effet papillon 131
Exercices 134

CHAPITRE 9
Vie et mort des objets 135

Question de mémoire 136
C++ : le programmeur est le seul maître à bord 145
En Java, C#, Python et PHP 5 : la chasse au gaspi 148
Exercices 154

CHAPITRE 10
UML 2 159

Diagrammes UML 2 161
Représentation graphique standardisée 162
Du tableau noir à l’ordinateur 163
Programmer par cycles courts en superposant
les diagrammes 164
Diagrammes de classe et diagrammes de séquence 165
Diagramme de classe 165
Les bienfaits d’UML 196
Diagramme de séquence 199
Exercices 205

CHAPITRE 11
Héritage 211

Comment regrouper les classes dans des superclasses 212
Héritage des attributs 213

Héritage ou composition ? 219
Économiser en rajoutant des classes ? 220
Héritage des méthodes 220
La recherche des méthodes dans la hiérarchie 229
Encapsulation protected 230
Héritage et constructeurs 231
Héritage public en C++ 237
Le multihéritage 238
Exercices 249

CHAPITRE 12
Redéfinition des méthodes 253

La redéfinition des méthodes 254
Beaucoup de verbiage mais peu d’actes véritables 255
Un match de football polymorphique 256
Exercices 288

CHAPITRE 13
Abstraite, cette classe est sans objet 299

De Canaletto à Turner 300
Des classes sans objet 300
Du principe de l’abstraction à l’abstraction syntaxique 301
Un petit supplément de polymorphisme 309
Exercices 313

CHAPITRE 14
Clonage, comparaison et assignation d’objets 325

Introduction à la classe Object 326
Décortiquons la classe Object 329
Test d’égalité de deux objets 331
Le clonage d’objets 336
Égalité et clonage d’objets en Python 339
Égalité et clonage d’objets en PHP 5 341
Égalité, clonage et affectation d’objets en C++ 343
En C#, un cocktail de Java et de C++ 353
Exercices 359

CHAPITRE 15

Interfaces 361

Interfaces : favoriser la décomposition et la stabilité 363
Java, C# et PHP 5 : interface via l'héritage 363
Les trois raisons d'être des interfaces 364
Les Interfaces dans UML 2 378
En C++ : fichiers .h et fichiers .cpp 379
Interfaces : du local à Internet 382
Exercices 382

CHAPITRE 16

Distribution gratuite d'objets : pour services rendus sur le réseau 387

Objets distribués sur le réseau : pourquoi ? 388
RMI (Remote Method Invocation) 391
Corba (Common Object Request Broker Architecture) 397
Rajoutons un peu de flexibilité à tout cela 404
Les services Web sur .Net 410
Exercices 420

CHAPITRE 17

Multithreading 423

Informatique séquentielle 425
Multithreading 427
Implémentation en Java 428
Implémentation en C# 430
Implémentation en Python 433
L'impact du multithreading sur les diagrammes de séquence UML 434
Du multithreading aux applications distribuées 435
Des threads équirépartis 435
Synchroniser les threads 437
Exercices 445

CHAPITRE 18

Programmation événementielle 449

Des objets qui s'observent 450
En Java 451
En C# : les délégués 454

En Python : tout reste à faire 462
Un feu de signalisation plus réaliste 465
Exercices 467

CHAPITRE 19
Persistance d’objets 469

Sauvegarder l’état entre deux exécutions 470
Simple sauvegarde sur fichier 471
Sauvegarder les objets sans les dénaturer : la sérialisation 478
Les bases de données relationnelles 483
Réservation de places de spectacles 495
Les bases de données relationnelles-objet 500
Les bases de données orientées objet 504
Linq 505
Exercices 509

CHAPITRE 20
Et si on faisait un petit flipper ? 511

Généralités sur le flipper et les GUI 513
Retour au Flipper 522

CHAPITRE 21
Les graphes 535

Le monde regorge de réseaux 536
Tout d’abord : juste un ensemble d’objets 538
Liste liée 539
La généricité en C++ 546
La généricité en Java 549
Passons aux graphes 555
Exercices 560

CHAPITRE 22
Petites chimie et biologie OO amusantes 565

Pourquoi de la chimie OO ? 566
Les diagrammes de classe du réacteur chimique 567
Quelques résultats du simulateur 581
La simulation immunologique en OO ? 583

CHAPITRE 23
Design patterns 589

Introduction aux design patterns 590
Les patterns « truc et ficelle » 592
Les patterns qui se jettent à l'OO 599

Index 613

Avant-propos

Dans les tout débuts de l'informatique, le fonctionnement « intime » des processeurs décidait toujours, en fin de compte, de la seule manière efficace de programmer un ordinateur. Alors que l'on acceptait tout programme comme une suite logique d'instructions, il était admis que l'organisation du programme et la nature même de ces instructions ne pouvaient s'éloigner de la façon dont le processeur les exécutait : pour l'essentiel, des modifications de données mémorisées, des déplacements de ces données d'un emplacement mémoire à un autre, et des opérations d'arithmétique et de logique élémentaire.

La mise au point d'algorithmes complexes, dépassant les simples opérations mathématiques et les simples opérations de stockage et de récupérations de données, obligea les informaticiens à effectuer un premier saut dans l'abstrait, en inventant un style de langage dit procédural, auquel appartiennent les langages Fortran, Cobol, Basic, Pascal, C... Ces langages permettaient à ces informaticiens de prendre quelques distances par rapport au fonctionnement intime des processeurs (en ne travaillant plus directement à partir des adresses mémoire et en évitant la manipulation directe des instructions élémentaires) et d'élaborer une écriture de programmes plus proches de la manière naturelle de poser et de résoudre les problèmes. Les codes écrits dans ces langages devenant indépendants en cela des instructions élémentaires propres à chaque type de processeur. Ces langages cherchaient à se positionner quelque part entre l'écriture des instructions élémentaires et l'utilisation tant du langage naturel que du sens commun. Il est incontestablement plus simple d'écrire : c = a + b qu'une suite d'instructions telles que : `"load a, reg1"`, `"load b, reg2"`, `"add reg3, reg1, reg2"`, `"move c, reg3"`, ayant pourtant la même finalité. Une opération de traduction automatique, dite de compilation, se charge alors de traduire le programme, écrit au départ dans ce nouveau langage, dans les instructions élémentaires, seules comprises par le processeur. Cette montée en abstraction permise par ces langages de programmation présente un double avantage : une facilitation d'écriture et de résolution algorithmique, ainsi qu'une indépendance accrue par rapport aux différents types de processeur existant aujourd'hui sur le marché.

Plus les problèmes à affronter gagnaient en complexité – comptabilité, jeux automatiques, compréhension et traduction des langues naturelles, aide à la décision, bureautique, conception et enseignement assistés, programmes graphiques, etc. –, plus l'architecture et le fonctionnement des processeurs semblaient contraignants, et plus il devenait vital d'inventer des mécanismes informatiques simples à mettre en œuvre, permettant une réduction de cette complexité et un rapprochement encore plus marqué de l'écriture des programmes des manières humaines de poser et de résoudre les problèmes.

Avec l'intelligence artificielle, l'informatique s'inspira de notre mode cognitif d'organisation des connaissances, comme un ensemble d'objets conceptuels entrant dans un réseau de dépendance et se structurant de manière taxonomique. Avec la systémique ou la bioinformatique, l'informatique nous révéla qu'un ensemble d'agents au fonctionnement élémentaire, mais s'influençant mutuellement, peut produire un comportement émergent d'une surprenante complexité. La complexité affichée par le comportement d'un système observé dans sa

globalité ne témoigne pas systématiquement d'une complexité équivalente lorsque l'attention est portée sur chacune des parties composant ce système et prise isolément. Dès lors, pour comprendre jusqu'à reproduire ce comportement par le biais informatique, la meilleure approche consiste en une découpe adéquate du système en ses parties et une attention limitée au fonctionnement de chacune d'entre elle.

Tout cela mis ensemble : la nécessaire distanciation par rapport au fonctionnement du processeur, la volonté de rapprocher la programmation du mode cognitif de résolution de problème, les percées de l'intelligence artificielle et de la bio-informatique, le découpage comme voie de simplification des systèmes apparement complexes, conduisit graduellement à un deuxième style de langage de programmation, un tout petit peu plus récent, bien que fêtant ses 45 ans d'existence (l'antiquité à l'échelle informatique) : les langages orientés objets, tels Simula, Smalltalk, C++, Eiffel, Java, C#, Delphi, Power Builder, Python et bien d'autres...

L'orientation objet en deux mots

À la différence de la programmation procédurale, un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Chaque objet se décrit par un ensemble d'attributs (partie statique) et un ensemble de méthodes portant sur ces attributs (partie dynamique). Certains de ces attributs étant l'adresse des objets avec lesquels les premiers collaborent, il leur est possible de déléguer certaines des tâches à leurs collaborateurs. Le tout s'opère en respectant un principe de distribution des responsabilités on ne peut plus simple, chaque objet s'occupant de ses propres attributs. Lorsqu'un objet exige de s'informer ou de modifier les attributs d'un autre, il charge cet autre de s'acquitter de cette tâche. Cette programmation est fondamentalement distribuée, modularisée et décentralisée. Pour autant qu'elle respecte également des principes de confinement et d'accès limité (dit d'encapsulation) que nous décrirons dans l'ouvrage, cette répartition modulaire a également l'insigne avantage de favoriser la stabilité des développements, en restreignant au maximum l'impact de modifications apportées au code au cours du temps. Ces impacts seront limités aux seuls objets qu'ils concernent et à aucun de leurs collaborateurs, même si le comportement de ces derniers dépend en partie des fonctionnalités affectées.

Ces améliorations, résultant de la prise de conscience des problèmes posés par l'industrie du logiciel ces dernières années, complexité accrue et stabilité dégradée, ont enrichi la syntaxe des langages objet.. Un autre mécanisme de modularisation inhérent à l'orienté objet est l'héritage qui permet à la programmation de refléter l'organisation taxonomique de notre connaissance en une hiérarchie de concepts du plus au moins général. À nouveau, cette organisation modulaire en objets génériques et plus spécialistes est à l'origine d'une simplification de la programmation, d'une économie d'écriture et de la création de zone de code aux modifications confinées. Tant cet héritage que la répartition des tâches entre les objets permet, tout à la fois, une décomposition plus naturelle des problèmes,une réutilisation facilitée des codes déjà existants, et une maintenance facilitée et allégée de ces derniers. L'orientation objet s'impose, non pas comme une panacée universelle, mais une évolution naturelle, au départ de la programmation procédurale, qui facilite l'écriture de programmes, les rendant plus gérables, plus compréhensibles, plus stables et réexploitables.

L'orienté objet inscrit la programmation dans une démarche somme toute très classique pour affronter la complexité de quelque problème qui soit : une découpe naturelle et intuitive en des parties plus simples. A fortiori, cette découpe sera d'autant plus intuitive qu'elle s'inspire de notre manière « cognitive » de découper la réalité qui nous entoure. L'héritage, reflet fidèle de notre organisation cognitive, en est le témoignage le plus éclatant. L'approche procédurale rendait cette découpe moins naturelle, plus « forcée ». Si de nombreux adeptes de la programmation procédurale sont en effet conscients qu'une manière incontournable de

simplifier le développement d'un programme complexe est de le découper physiquement, ils souffrent de l'absence d'une prise en compte naturelle et syntaxique de cette découpe dans les langages de programmation utilisés. Dans un programme imposant, l'OO permet de tracer les pointillés que les ciseaux doivent suivre là où il semble le plus naturel de les tracer : au niveau du cou, des épaules ou de la ceinture, et non pas au niveau des sourcils, des biceps ou des mollets. De surcroît, cette pratique de la programmation incite à cette découpe suivant deux dimensions orthogonales : horizontalement, les classes se déléguant mutuellement un ensemble de services, verticalement, les classes héritant entre elles d'attributs et de méthodes installés à différents niveaux d'une hiérarchie taxonomique. Pour chacune de ces dimensions, reproduisant fidèlement nos mécanismes cognitifs de conceptualisation, en plus de simplifier l'écriture des codes, il est important de faciliter la récupération de ces parties dans de nouveaux contextes et d'assurer la robustesse de ces parties aux changements survenus dans d'autres. Un code OO, idéalement, sera aussi simple à créer qu'à maintenir, récupérer et faire évoluer.

Il est parfaitement inconséquent d'opposer le procédural à l'OO car, in fine, toute programmation des méthodes (c'est-à-dire la partie active des classes et des objets) reste totalement tributaire des mécanismes procéduraux. On y rencontre des variables, des arguments, des boucles, des arguments de fonction, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils procédurales. L'OO ne permet en rien de faire l'économie du procédural, simplement, il complémente celui-ci, en lui superposant un système de découpe plus naturel et facile à mettre en œuvre. Il n'est guère surprenant que la plupart des langages procéduraux comme le C, Cobol ou, plus récemment, PHP, se soient relativement aisément enrichi d'une couche dite OO sans que cette addition ne remette sérieusement en question l'existant procédural. Cependant, l'impact de cette couche additionnelle ne se limite pas à quelques structures de données supplémentaires afin de mieux organiser les informations manipulées par le programme. Il va bien au-delà. C'est toute une manière de concevoir un programme et la répartition de ses parties fonctionnelles qui est en jeu. Les fonctions et les données ne sont plus d'un seul tenant mais éclatées en un ensemble de modules reprenant, chacun, une sous-partie de ces données et les seules fonctions qui les manipulent. Il faut réapprendre à programmer en s'essayant au développement d'une succession de micro-programmes et au couplage soigné et réduit au minimum de ces micro-programmes. En substance, la programmation OO pourrait reprendre à son compte ce slogan devenu très célèbre parmi les adeptes des courants altermondialistes : « agir localement, penser globalement ». Se pose alors la question de stratégie pédagogique, question très controversée dans l'enseignement de l'informatique aujourd'hui, sur l'ordre chronologique à donner au procédural et à l'OO. De nombreux enseignants de la programmation, soutenus en cela par de très nombreux manuels de programmation, considèrent qu'il faut d'abord passer par un enseignement intensif et une maîtrise parfaite du procédural, avant de faire le grand saut vers l'OO. Quinze années d'enseignement de la programmation à des étudiants de tout âge et de toute condition (de 7 à 77 ans, issus des sciences humaines ou exactes) nous ont convaincu qu'il n'y a aucun ordre à donner. De même qu'historiquement, l'OO est né quasiment en même temps que le procédural et en complément de celui-ci, l'OO doit s'enseigner conjointement et en complément du procédural. Il faut enseigner les instructions de contrôle en même temps que la découpe en classe. Tout comme un cours de cuisine s'attardant sur quelques ingrédients culinaires très particulier parallèlement à la manière dont ces ingrédients doivent s'harmoniser, ou un cours de mécanique automobile se focalisant sur quelques pièces ou mécanismes en particulier en même temps que le plan et le fonctionnement d'ensemble, l'enseignement de la programmation doit mélanger à loisir la perception « micro » des mécanismes procéduraux à la vision « macro » de la découpe en objets. Aujourd'hui, tout projet informatique de dimension conséquente débute par une analyse des différentes classes qui le constituent. Il faut aborder l'enseignement de la programmation tout comme débute la prise en charge de ce type de projet, en enseignant au plus vite la manière dont ces classes et les objets qui en résultent opèrent à l'intérieur d'un programme.

L’orienté objet s’est trouvé à l’origine ces dernières années, compétition oblige, d’une explosion de technologies différentes, mais toutes intégrant à leur manière les mécanismes de base de l’OO : classes, objets, envois de messages, héritage, encapsulation, polymorphisme... Ainsi sont apparus une multitude de langages de programmation, qui intègrent ces mécanismes de base à leur manière, à partir d’une syntaxe dont les différences sont soit purement cosmétiques, soit légèrement plus subtiles. Ils sont autant de variations sur le ou les thèmes créés par leurs trois principaux précurseurs : Simula, Smalltalk et C++.

L’OO a également permis de repenser trois des chapitres les plus important de l’informatique de ces deux dernières décennies. Tout d’abord, le besoin d’une méthode de modélisation graphique débouchant sur un niveau d’abstraction encore supplémentaire (on ne programme plus, on dessine un ensemble de diagrammes, le code étant généré automatiquement à partir de ceux-ci) (rôle joué par UML 2) ; ensuite, les applications informatiques distribuées (on ne parlera plus d’applications distribuées mais d’objets distribués, et non plus d’appels distants de procédures mais d’envoi de messages à travers le réseau) ; enfin, le stockage des données qui doit maintenant compter avec les objets. Chaque fois, plus qu’un changement de vocabulaire, un changement de mentalité sinon de culture s’impose.

Aujourd’hui, force est de constater que l’OO constitue un sujet d’une grande attractivité pour tous les acteurs de l’informatique. Microsoft a développé un nouveau langage informatique purement objet, C#, a très intensément contribué au développement d’un système d’informatique distribuée, basé sur des envois de messages d’ordinateur à ordinateur, les services web, et a plus récemment proposé un nouveau langage d’interrogation des objets, LINQ, qui s’interface naturellement avec le monde relationnel et le monde XML. Tous les langages informatiques intégrés dans sa nouvelle plate-forme de développement, Visual Studio .Net (aux dernières nouvelles, ils seraient 22), visent à une uniformisation (y compris les nouvelles versions de Visual Basic et Visual C++) en intégrant les mêmes briques de base de l’OO. Aboutissement considérable s’il en est, il devient très simple de faire communiquer ou hériter entre elles des classes écrites dans des langages différents. Quelques années auparavant, Sun avait créé Java,une création déterminante car à l’origine de ce nouvel engouement pour une manière de programmer qui pourtant existait depuis toujours sans que les informaticiens dans leur ensemble en reconaissent l’utilité et la pertinence. Depuis, en partant de son langage de prédilection, Sun à créé RMI, Jini, et sa propre version des services Web, tous basés sur les technologies OO. Ces mêmes services Web font l’objet de développements tout autant aboutis chez HP ou IBM. À la croisée de Java et du Web, originellement, la raison sinon du développement de Java du moins de son succès, on découvre une importante panoplie d’outils de développement et de conception de sites Web dynamiques.

IBM et Borland, en rachetant respectivement Rational et Together, mènent la danse en matière d’outil d’analyse du développement logiciel, avec la mise au point de puissants environnements UML, technologie OO comme il se doit. Au départ de développements chez IBM (qui soutient et parie sur Java plus encore que SUN ne le fait), la plate-forme logicielle Eclipse est sans doute, à ce jour, l’aventure Open Source la plus aboutie en matière d’OO. Comme environnement de développement Java, Eclipse est aujourd’hui le plus prisé et le plus usité et gagne son pari « d’éclipser » tous les autres. Borland a rendu Together intégrable tant dans Visual Studio.Net que dans Eclipse comme outil de modélisation UML synchronisant au mieux et au plus la programmation et la réalisation des diagrammes UML. Enfin, l’OMG, organisme de standardisation du monde logiciel, n’a pas comme lettre initiale de son acronyme la lettre O pour rien. UML et Corba sont ses premières productions : la version OO de l’analyse logicielle et la version OO de l’informatique distribuée. Cet organisme plaide de plus en plus pour un développement informatique détaché des langages de programmation ainsi que des plates-formes matérielles, par l’utilisation intensive des diagrammes UML. Au départ de ces mêmes diagrammes, les codes seraient générés automatiquement dans un langage choisi et en adéquation avec

la technologie voulue. Par le nouveau saut dans l'abstraction qu'il autorise, UML se profilerait comme le langage de programmation de demain. Il jouerait à ce titre le même rôle que jouèrent les langages de programmation au temps de leur apparition, en relèguant ceux-ci à la même place que le langage assembleur auquel ils se sont substitués jadis : un pur produit de traduction automatisée. Au même titre qu'Unix pour les développements en matière de système d'exploitation, l'OO donc apparaît comme le point d'orgue et de convergence de ce qui se fait de plus récent en matière de langages et d'outils de programmation.

Objectifs de l'ouvrage

Toute pratique économe, fiable et élégante de Java, C++, C#, Python, .Net ou UML requiert, pour débuter, une bonne maîtrise des mécanismes de base de l'OO. Et, pour y pourvoir, rien de mieux que d'expérimenter les technologies OO dans ces différentes versions, comme un bon conducteur qui se sera frotté à plusieurs types de véhicule, un bon skieur à plusieurs styles de ski et un guitariste à plusieurs modèles de guitare.

Plutôt qu'un voyage en profondeur dans l'un ou l'autre de ces multiples territoires, ce livre vous propose d'explorer plusieurs d'entre eux, mais en tentant à chaque fois de dévoiler ce qu'ils recèlent de commun. Car ce sont ces ressemblances qui constituent en dernier ressort les briques fondamentales de l'OO, matière de base, qui se devrait de perdurer encore de nombreuses années, y compris sous de nouveaux déguisements. Nous pensons que la mise en parallèle de C++, de Java, C#, Python, PHP 5 et UML est une voie privilégiée pour l'extraction de ces mécanismes de base.

Il nous a paru pour cette raison indispensable de discuter et comparer la façon dont ces cinq langages de programmation gèrent, par exemple, l'occupation mémoire par les objets ou leur manière d'implémenter le polymorphisme, pour en comprendre, *in fine*, toute la problématique et les subtilités indépendamment de l'une ou l'autre implémentation. Rajoutez une couche d'abstraction, ainsi que le permet UML, et cette compréhension ne pourra s'en trouver que renforcée. Chacun de ces cinq langages offrent des particularités amenant les praticiens de l'un ou l'autre à le prétendre mordicus supérieur aux autres : la puissance du C++, la compatibilité Windows et l'intégration XML de C#, l'anti-Microsoft et le leadership de Java en matière de développement Web, les vertus pédagogiques et l'aspect « scripting » de Python, le succès incontestable de PHP 5 pour la mise en place de solution Web dynamique et capable de s'interfacer aisément avec les bases de données. Nous nous désintéresserons ici complètement de ces guerres de religion (qui partagent avec les guerres de langages informatiques pas mal d'irrationalité), a fortiori car notre projet pédagogique nous conduit bien davantage à nous pencher sur ce qui les réunit plutôt que ce qui les différencie. C'est leur multiplicité qui a présidé à cet ouvrage et qui en fait, nous l'espérons, son originalité. Nous n'allons pas nous en plaindre et défendons en revanche l'idée que le choix définitif de l'un ou l'autre de ces langages dépend davantage d'habitude, d'environnement professionnel ou d'enseignement, de questions sociales et économiques et surtout de la raison concrète de cette utilisation (pédagogie, performance machine, adéquation Web ou base de données, …). De plus, le succès d'UML, assimilable à un langage universel OO à l'intersection de tous les autres et automatiquement traduisible dans chacun, ou des efforts, tels ceux de Microsoft, d'homogénéisation des langages OO, rend ces discordes quelque peu obsolètes et un peu dérisoires, tant il va devenir facile de passer de l'un à l'autre. Enfin, nous souhaitions que cet ouvrage, tout en étant suffisamment détaché de toutes technologies, couvre l'essentiel des problèmes posés par la mise en œuvre des objets en informatique, y compris le problème de leur stockage sur le disque dur et leur interfaçage avec les bases de données, de leur fonctionnement en parallèle, et leur communication à travers Internet. Un ouvrage donc qui découvrirait l'OO de très haut, ce qui lui permet évidemment de balayer très large, et qui accepte ce faisant de perdre un peu en précision, perte dont nous il apparaît nécessaire de mettre en garde le lecteur.

Plan de l’ouvrage

Les 23 chapitres de ce livre peuvent se répartir en cinq grandes parties.

Le premier chapitre constitue une partie en soi car il a pour importante mission d’introduire aux briques de base de la programmation orientée objet, sans aucun développement technique : une première esquisse, teintée de sciences cognitives, et toute en intuition, des éléments essentiels de la pratique OO.

La deuxième partie intègre les quatorze chapitres suivants. Il s’agit pour chacun d’entre eux de décrire, plus techniquement cette fois, ces briques de base que sont : objet, classe (chapitres **2** et **3**), messages et communication entre objets (chapitres **4**, **5** et **6**), encapsulation (chapitres **7** et **8**), gestion mémoire des objets (chapitre **9**), modélisation objet (chapitre **10**), héritage et polymorphisme (chapitres **11** et **12**), classe abstraite (chapitre **13**), clonage et comparaison d’objets (chapitre **14**), interface (chapitre **15**).

Chacune de ces briques est illustrée par des exemples en Java, C#, C++,Python, PHP 5 et UML. Nous y faisons le pari que cette mise en parallèle est la voie la plus naturelle pour la compréhension des mécanismes de base : extraction du concept par la multiplication des exemples.

La troisième partie reprend, dans le droit fil des ouvrages dédiés à l’un ou l’autre langage objet, des notions jugées plus avancées : les objets distribués, Corba, RMI, Services Web (chapitre **16**), le multithreading ou programmation parallèle (ou concurrentielle, chapitre **17**), la programmation événementielle (chapitre **18**) et enfin la sauvegarde des objets sur le disque dur, y compris l’interfaçage entre les objets et les bases de données relationnelles (chapitre **19**). Là encore, le lecteur se trouvera le plus souvent en présence de plusieurs versions dans les quatre langages de ces mécanismes.

La quatrième partie décrit plusieurs projets de programmation objet totalement aboutis, tant en UML qu’en Java. Elle inclut d’abord le chapitre **20**, décrivant la modélisation objet d’un petit flipper et les problèmes de conception orientée objet que cette modélisation pose. Le chapitre **21**, lié au chapitre 22, décrit la manière dont les objets peuvent s’organiser en liste liée ou en graphe, mode de mise en relation et de regroupement des objets que l’on retrouve abondamment dans toute l’informatique. Le chapitre **22** marie la chimie et la biologie à la programmation OO. Il contient tout d’abordla programmation d’un réacteur chimique générant de nouvelles molécules à partir de molécules de base, et ce, tout en suivant à la trace l’évolution de la concentration des molécules dans le temps. La chimie – une chimie élémentaire acquise bien avant l’université – nous est apparue être une plate-forme pédagogique idéale pour l’assimilation des concepts objets. Nous ne surprendrons personne en affirmant que les atomes et les molécules sont deux types de composants chimiques, et que les secondes sont composées des premiers. Dans ce chapitre, nous traduisons ces connaissances en UML et en Java. Dans la suite de la chimie, nous proposons aussi dans le chapitre une simulation élémentaire du système immunitaire, comme nouvelle illustration de combien l’informatique OO se prête facilement à la reproduction informatisée des concepts de science naturelle, tels ceux que l’on rencontre en chimie ou en biologie.

Enfin la dernière partie, se ramène au seul dernier chapitre, le chapitre 23, dans lequel sont présentés un ensemble de recettes de conception OO, solutionnant de manière fort élégante un ensemble de problèmes récurrents dans la réalisation de programme OO. Ces recettes de conception, dénommées Design Pattern, sont devenues fort célèbres dans la communauté OO. Leur compréhension accompagne une bonne maîtrise des principes OO et s’inscrit dans la suite logique de l’enseignement des briques et des mécanismes de base de l’OO. Elle fait souvent la différence entre l’apprenti et le compagnon parmi les programmeurs OO. Nous les illustrons en partie sur le flipper,la chimie et la biologie des chapitres précédents.

À qui s'adresse ce livre ?

Cet ouvrage ayant pour objet de traiter de nombreuses technologies, nul doute qu'il est destiné à être lu par un public assez large. En clair, il s'adresse à tous les adeptes de chacune de ces technologies, industriels, enseignants et étudiants, qui pourront le confronter utilement à l'état de l'art en la matière. La vocation première de cet ouvrage n'en reste pas moins une initiation à la programmation orientée objet, prérequis indispensable à l'assimilation de nombreuses autres technologies.

Ce livre sera un compagnon d'étude utile et, nous l'espérons, enrichissant pour les étudiants qui comptent la programmation objet dans leur cursus d'étude (et toutes technologies s'y rapportant : Java, C++, C#, Python, PHP, Corba, RMI, Services Web, UML). Il devrait les aider, le cas échéant, à évoluer de la programmation procédurale à la programmation objet, pour aller ensuite vers toutes les technologies s'y rapportant.

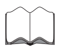
Nous ne pensons pas, en revanche, que ce livre peut seul prétendre à une même porte d'entrée dans le monde de la programmation tout court. Comme dit précédemment, nous pensons qu'il est idéal d'aborder les mécanismes OO en même temps que procéduraux. Pour des raisons évidentes de place et car les librairies informatiques déjà en regorgent, nous avons fait l'impasse d'un enseignement de base des mécanismes procéduraux : variables, boucles, instructions conditionnelles, éléments fondamentaux et compagnons indispensables à l'assimilation de l'OO. Nous pensons, dès lors, que ce livre sera plus facile à aborder pour des lecteurs ayant déjà un peu de pratique de la programmation dite procédurale, et ce, dans un quelconque langage de programmation. Aujourd'hui, l'informatique est un sujet si vaste, existant à tant de niveaux d'abstraction, et pour tant de raisons différentes, qu'il n'est pas étonnant qu'il faille l'aborder muni de plusieurs guides. Ce livre en est un. Il n'a rien d'exhaustif, ne se spécialise dans aucune des technologies évoquées, mais fournit les bases nécessaires à l'assimilation d'un grand nombre d'entre elles et de celles à venir.

1

Principes de base : quel objet pour l'informatique ?

Ce chapitre a pour but une introduction aux briques de base de la conception et de la programmation orientée objet (OO). Il s'agit pour l'essentiel des notions d'objet, de classe, de message et d'héritage. À ce stade, aucun approfondissement technique n'est effectué. Les quelques bouts de code seront écrits dans un pseudo langage très proche de Java. De simples petits exercices de pensée permettent une mise en bouche, toute en intuition, des éléments essentiels de la pratique OO.

Sommaire : Introduction à la notion d'objet — Introduction à la notion et au rôle du référent — L'objet dans sa version passive et active — Introduction à la notion de classe — Les interactions entre objets — Introduction aux notions d'héritage et de polymorphisme



Doctus — Tu as l'air bien remonté, aujourd'hui !

Candidus — Je cherche un objet, mon vieux ! C'est l'objet que je cherche partout.

Doc. — Ce n'est pourtant pas ce qui manque… Tiens, prends donc ma valise…

Cand. — Non, je cherche un objet autrement plus encombrant… C'est ce sacré objet logiciel dont tout le monde parle. Il me fait penser au Yéti… Je me demande si quelqu'un en a vraiment rencontré un…

Doc. — Quelle idée, il n'a rien d'aussi mystérieux notre objet.. Il s'agit simplement de petits soldats qui vont nous libérer de bien des contraintes du monde procédural.

Cand. — Justement ! Dis-moi ce qu'est cette guerre Procédural contre Objet.

Doc. — Au commencement… il y avait l'ordinateur, avec toutes ses faiblesses de nouveau-né. C'est nous qui étions à son service pour le pouponner. Il fallait être sacrément malin pour en tirer quelque chose.

Cand. — Et maintenant il a grandi et je parie qu'il veut jouer avec des petits objets.

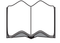
Doc. — Il a effectivement pris du plomb dans la tête et il comprend beaucoup mieux ce qu'on attend de lui. On peut lui parler en adulte, lui expliquer les choses d'une façon plus structurée…

Cand. — …Veux-tu dire qu'il serait capable de comprendre ce que nous voulons sous forme de spécification ?

Doc. — Doucement ! Je dis simplement que nous ne passerons plus tout notre temps à considérer ce que nos processeurs attendent pour faire le travail demandé. C'est la première des étapes que nous avons déjà franchies.

Cand. — Quelles sont les autres étapes ?

Doc. — Et bien notre bébé est aujourd'hui capable de manipuler lui-même les informations qu'on lui confie. Il a ses propres méthodes d'utilisation et de rangement. Il ne veut même plus qu'on touche à ses jouets.



Un rapide coup d'œil par la fenêtre et nous apercevons… des voitures, des passants, des arbres, un immeuble, un avion… Cette simple perception est révélatrice d'un ensemble de mécanismes cognitifs des plus subtils, dont la compréhension est une porte d'entrée idéale dans le monde de l'informatique orientée objet. En effet, pourquoi n'avoir pas cité « l'air ambiant », la « température », la « bonne ambiance » ou, encore, « la lumière du jour », que l'on perçoit tout autant ? Pourquoi les premiers se détachent-ils de cette toile de fond parcourue par nos yeux ?

Tout d'abord, leur structure est singulière, compliquée, ils présentent une forme alambiquée, des dimensions particulières, parfois une couleur uniforme et distincte du décor qui les entoure. Nous dirons que chacun se caractérise par un ensemble « d'attributs » structuraux, prenant pour chaque objet une valeur particulière : une des voitures est rouge, plutôt longue, ce passant est grand, assez vieux, courbé, etc. Ces attributs structuraux – et leur présence conjointe dans les objets – sont la raison première de l'attrait perceptif qu'ils exercent. C'est aussi la raison de la segmentation et de la nette séparation perceptive qui en résulte, car si les voitures et les arbres se détachent en effet de l'arrière plan, nous ne les confondons en rien.

Le trio <entité, attribut, valeur>

Nous avons l'habitude de décrire le monde qui nous entoure à l'aide de ce trio que les informaticiens se plaisent à nommer : <entité, attribut, valeur>, par exemple : <voiture, couleur, rouge>, <voiture, marque, peugeot>, <passant, taille, grande>, <passant, âge, 50>. Si ce sont les entités et non pas leurs attributs qui vous sautent aux yeux, c'est bien parce que chacune de ces entités ou objets, la voiture et le passant, se caractérise par plusieurs de ces attributs, couleur, âge, taille, prenant une valeur particulière, uniforme « sur tout l'objet ».

L'objet est perçu, de fait, car il est au croisement de ces différents attributs. Il naît à partir de leur rencontre. Les attributs en tant que tels ne sont pas des objets, puisqu'ils servent justement à la caractérisation de ces objets, à les faire exister et à les rendre prégnants.

La nature des attributs est telle qu'ils se retrouvent attribut d'un nombre important d'objets, pourtant très différents. La voiture a une taille comme le passant. L'arbre a une couleur comme la voiture. Le monde des attributs est beaucoup moins diversifié que le monde des objets. C'est une des raisons qui nous permettent de regrouper les objets en classes et les classes en différentes sous-classes, comme nous le découvrirons plus tard. Que ce soit comme résultat de nos perceptions ou dans notre pratique langagière, les attributs et les objets jouent des rôles très différents. Les attributs structurent nos perceptions et ils servent, par exemple, sous forme d'adjectifs, à qualifier les noms qui les suivent ou les précèdent. La première conséquence de cette simple faculté de découpage cognitif sur l'informatique d'aujourd'hui est la suivante :

Objets, attributs, valeurs

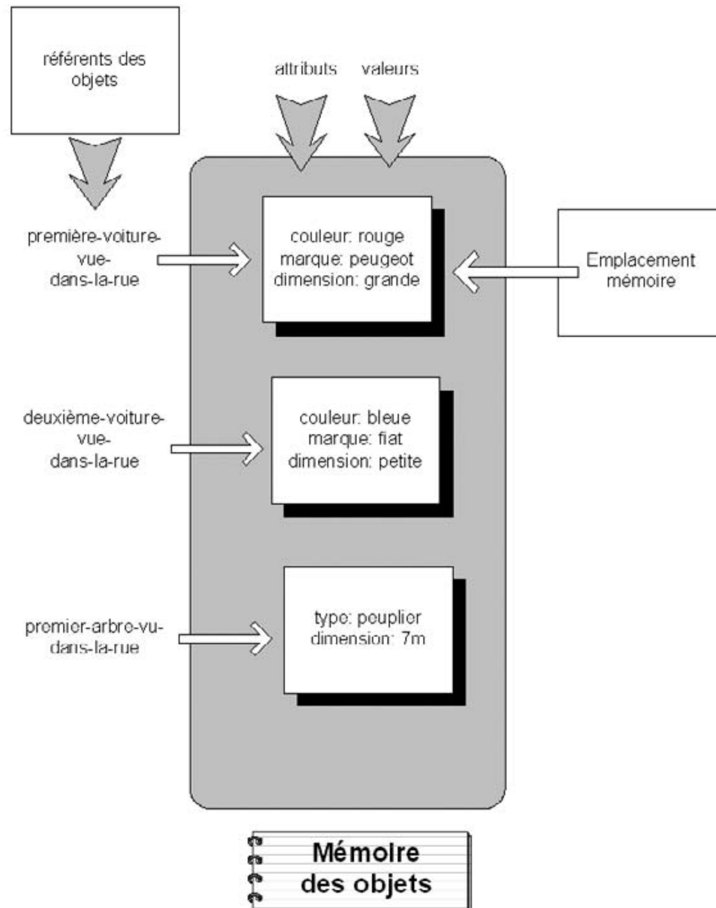
Il est possible dans tous les langages informatiques de stocker et de manipuler des objets en mémoire, comme autant d'ensembles de couples attribut/valeur.

Stockage des objets en mémoire

Dans la figure qui suit, nous voyons apparaître ces différents objets dans la mémoire de l'ordinateur. Chacun occupe un espace mémoire qui lui est propre et alloué lors de sa création. De façon à se faciliter la vie, les informaticiens ont admis un ensemble de « types primitifs » d'attribut, dont ils connaissent à l'avance la taille requise pour encoder la valeur.

Figure 1-1

Les objets informatiques et leur stockage en mémoire.



Il s'agit, par exemple, de types comme *réel* qui occupera 64 bits d'espace (dans le codage des réels en base 2 et selon une standardisation reconnue) ou *entier*, qui en occupera 32 (là encore par sa traduction en base 2), ou finalement *caractère* qui occupera 16 bits dans le format « unicode » (qui code ainsi chacun des caractères de la majorité des écritures répertoriées et les plus pratiquées dans le monde). Dans notre exemple, les dimensions seraient typées en tant qu'entier ou réel. Tant la couleur que la marque pourraient se réduire à une valeur numérique (ce qui ramènerait l'attribut à un entier) choisie parmi un ensemble fini de valeurs possibles, indiquant chacune une couleur ou une marque. Dès lors, le mécanisme informatique responsable du stockage de l'objet « saura », à la simple lecture structurelle de l'objet, quel est l'espace mémoire requis par son stockage.

> **La place de l'objet en mémoire**
> Les objets seront structurellement décrits par un premier ensemble d'attributs de type primitif, tels qu'entier, réel ou caractère, qui permettra, précisément, de déterminer l'espace qu'ils occupent en mémoire.

Types primitifs

À l'aide de ces types primitifs, le stockage en mémoire de ces différents objets se transforme comme reproduit dans la figure 1-2. Ce mode de stockage de données est une caractéristique récurrente en informatique, présent dans pratiquement tous les langages de programmation, et se retrouvant dans les bases de données dites relationnelles. Dans ces bases de données, chaque objet devient un enregistrement. Les voitures sont stockées à l'aide de leurs couples attribut/valeur dans des bases encodant des voitures, et gérées, par exemple, par un concessionnaire automobile (comme montré à la figure 1-3).

Figure 1-2
Les objets avec leur nouveau mode de stockage où chaque attribut est d'un type dit « primitif » ou « prédéfini », comme entier, réel, caractère…

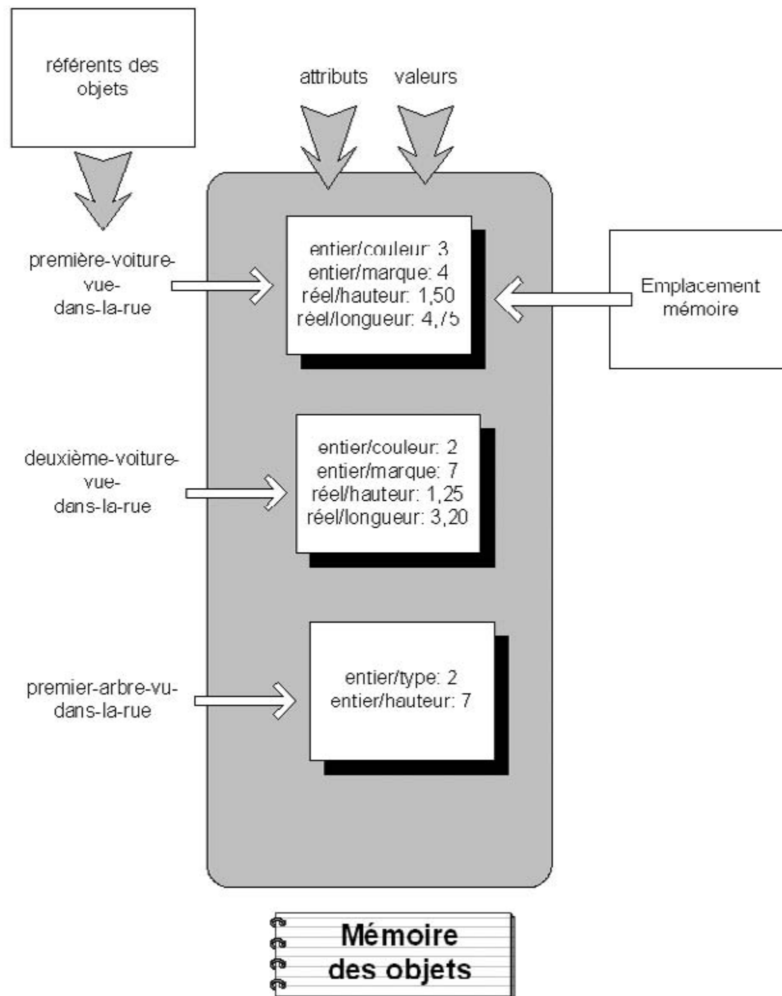


Figure 1-3
Table d'une base de données relationnelle de voitures, avec quatre attributs et six enregistrements.

Marque	Modèle	Série	Numéro
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

Bases de données relationnelles

Il s'agit du mode de stockage des données sur support permanent le plus répandu en informatique. Les données sont stockées en tant qu'enregistrement dans des tables, par le biais d'un ensemble de couples attribut/valeur dont une clé primaire essentielle à la singularisation de chaque enregistrement. Des relations sont ensuite établies entre les tables par un mécanisme de jonction entre la clé primaire de la première table et la clé dite étrangère de celle à laquelle on désire la relier. Le fait, par exemple, qu'un conducteur puisse posséder plusieurs voitures se traduit en relationnel par la présence dans la table voiture d'une clé étrangère qui reprend les valeurs de la clé primaire présente dans la table des conducteurs. La disparition de ces clés dans la pratique OO fait de la sauvegarde des objets dans ces tables un problème épineux de l'informatique d'aujourd'hui, comme nous le verrons au chapitre 19.

Les arbres, quant à eux, chacun également avec leurs couples attribut/valeur, s'enregistrent dans des bases de données gérées par un botaniste. Cette façon de procéder n'a rien de novateur et n'est en rien à l'origine de cette pratique informatique désignée comme orientée objet. La simple opération de stockage et manipulation d'objet en soi et pour soi n'est pas ce qui distingue fondamentalement l'informatique orientée objet de celle désignée comme « procédurale » ou « fonctionnelle ». Nous la retrouvons dans pratiquement tous les langages informatiques. Patience ! Nous allons y venir…

De tout temps également, les mathématiciens, physiciens, ou autres scientifiques, ont manipulé des objets mathématiques caractérisés par un ensemble de couples attribut/valeur. Ainsi, un point dans un espace à trois dimensions se caractérise par les valeurs réelles prises par ses attributs x,y,z. Lorsque ce point bouge, on peut y adjoindre trois nouveaux attributs pour représenter sa vitesse. Il en est de même pour une espèce animale, caractérisée par le nombre de ses représentants, d'un atome, caractérisé par son nombre atomique, et d'une molécule par le nombre d'atomes qui la composent et par sa concentration au sein d'un mélange chimique. Même chose pour la santé économique d'un pays, caractérisée par le PIB par habitant, la balance commerciale ou le taux d'inflation.

Le référent d'un objet

Observons à nouveau les figures 1-1 et 1-2. Chaque objet est nommé et ce nom doit être unique. Le nom de l'objet est son seul et unique identifiant. Comme c'est en le nommant que nous accédons à l'objet, il est clair que ce nom ne peut être partagé par plusieurs objets. En informatique, le nom correspondra de manière univoque à l'adresse physique de l'objet en mémoire. Rien de plus unique qu'une adresse, sauf à supposer que deux objets puissent occuper le même espace mémoire. Pas d'inquiétude pour eux, nos objets ont les moyens de ne pas squatter la mémoire ! Le nom « première-voiture-vue-dans-la-rue » est en fait une variable informatique, que nous appellerons *référent* par la suite, stockée également en mémoire, mais dans un espace dédié

uniquement aux noms symboliques. À cette variable on assigne comme valeur l'adresse physique de l'objet que ce nom symbolique désigne. En général, dans la plupart des ordinateurs aujourd'hui, l'adresse mémoire se compose de 32 bits, ce qui permet de stocker jusqu'à 2^{32} informations différentes. Un référent est donc une variable informatique particulière, associée à un nom symbolique, codée sur 32 bits, et contenant l'adresse physique d'un objet informatique.

Espace mémoire

Le référent contient l'adresse physique de l'objet, codée sur 32 bits dans la plupart des ordinateurs aujourd'hui. Le nombre d'espaces mémoire disponibles est lié à la taille de l'adresse de façon exponentielle. Ces dernières années, de plus en plus de processeurs, tant chez Sun, Apple ou Intel, ont fait le choix d'une architecture à 64 bits, ce qui implique notamment une révision profonde de tous les mécanismes d'adressage dans les systèmes d'exploitation. Depuis, les informaticiens peuvent voir l'avenir avec confiance et se sentir à l'aise pour des siècles et des siècles face à l'immensité de l'espace d'adressage qui s'ouvre à eux. Celui-ci devient de 2^{64}, soit 18.446.744.073.709.551.616 octets. Excusez du peu. Aura-t-on jamais suffisamment de données et de programmes à y installer ?

Référent vers un objet unique

Le nom d'un objet informatique, ce qui le rend unique, est également ce qui permet d'y accéder physiquement. Nous appellerons ce nom le « référent de l'objet ». L'information reçue et contenue par ce référent n'est rien d'autre que l'adresse mémoire où cet objet se trouve stocké.

Plusieurs référents pour un même objet

Un même objet peut-il porter plusieurs noms ? Plusieurs référents, qui contiennent tous la même adresse physique, peuvent-ils désigner en mémoire un seul et même objet ? Oui, s'il est nécessaire de nommer, donc d'accéder à l'objet, dans des contextes différents et qui s'ignorent mutuellement.

Dans la vie courante, rien n'interdit à plusieurs personnes, tout en désignant le même objet, de le nommer de manière différente. Le livre que vous vous devez d'acheter en vingt exemplaires pour le faire connaître autour de vous, le livre dont tous les informaticiens raffolent, le best-seller de l'année, le chef-d'œuvre absolu, autant de référents différents pour désigner cet unique ouvrage que vous tenez précieusement entre les mains. Les noms des objets seront distincts, car utilisés dans des contextes distincts. C'est aussi faisable en informatique orientée objet, grâce à ce mécanisme puissant et souple de référence informatique, dénommé *adressage indirect* par les informaticiens qui permet, sans difficulté, d'offrir plusieurs voies d'accès à un même objet mémoire. Comme la pratique orienté objet s'accompagne d'une découpe en objets et que chacun d'entre eux peut être sollicité par plusieurs autres qui « s'ignorent » entre eux, il est capital que ces derniers puissent désigner ce premier à leur guise en lui donnant un nom plus conforme à l'utilisation qu'ils en feront.

Adressage indirect

C'est la possibilité pour une variable, non pas d'être associée directement à une donnée, mais plutôt à une adresse physique d'un emplacement contenant, lui, cette donnée. Il devient possible de différer le choix de cette adresse pendant l'exécution du programme, tout en utilisant naturellement la variable. Et plusieurs de ces variables peuvent alors pointer vers un même emplacement. Une telle variable est dénommée un pointeur, en C et C++.

> **Plusieurs référents pour un même objet**
>
> Dans le cours de l'écriture d'un programme orienté objet, on accédera couramment à un même objet par plusieurs référents, générés dans différents contextes d'utilisation. Cette multiplication des référents sera un élément déterminant de la gestion mémoire associée à l'objet. On acceptera à ce stade-ci qu'il est utile qu'un objet séjourne en mémoire tant qu'il est possible de le référer. Sans référent un objet est bon pour la poubelle puisque inaccessible. Vous êtes mort, je jour où vous n'êtes même plus un numéro dans aucune base de données.

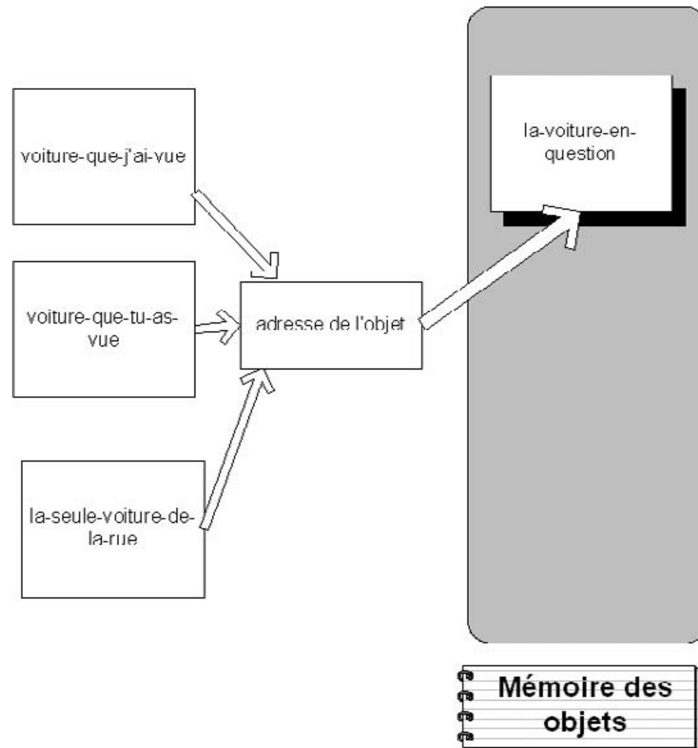


Figure 1-4
Plusieurs référents désignent un même objet grâce au mécanisme informatique d'adressage indirect.

Nous verrons dans la section suivante qu'un attribut peut servir de référent vers un autre objet, et qu'il y a là un mécanisme idéal pour permettre à ces deux objets de communiquer, par envoi de messages, du premier vers le deuxième. Mais n'allons pas trop vite en besogne…

L'objet dans sa version passive

L'objet et ses constituants

Voyons plus précisément ce qui amène notre perception à privilégier certains objets plutôt que d'autres. Certains d'entre eux se révèlent être une composition subtile d'autres objets, objets évidemment tout aussi présents que les premiers, et que, pourtant, il ne vous est pas venu à l'idée de citer lors de notre première démonstration.

Vous avez dit « la voiture » et non pas « la roue de la voiture » ou « sa portière », vous avez dit « l’arbre », et non pas « la branche » ou « le tronc de l’arbre ». De nouveau, c’est l’agrégat qui vous saute aux yeux, et non pas toutes ses parties. Vous savez pertinemment que l’objet « voiture » ne peut fonctionner en l’absence de ses objets « roues » ou de son objet « moteur ». Néanmoins, pour citer ce que vous observiez, vous avez fait l’impasse sur les différentes parties constitutives des objets relevés.

Première distinction : ce qui est utile à soi et ce qui l’est aux autres

L’orienté objet, pour des raisons pratiques que nous évoquerons par la suite, encourage à séparer, dans la description de tout objet, la partie utile pour tous les autres objets qui y recouront de la partie nécessaire à son fonctionnement propre. Il faut séparer physiquement ce que les autres objets doivent savoir d’un objet donné, afin de solliciter ses services, et ce que ce dernier requiert pour son fonctionnement, c’est-à-dire la mise en œuvre de ces mêmes services.

Objet composite

En tant que banal utilisateur de l’objet « voiture », vous vous préoccuperez des roues et du moteur comme de l’an 40. À moins que vous en ayez un besoin direct et incontournable, le garagiste se chargera bien tout seul de vous ruiner ! Que les objets s’organisent entre eux, en composite et composant, est une donnée de notre réalité que les informaticiens ont jugé important de reproduire. Comme indiqué dans la figure suivante, un objet stocké en mémoire peut être placé à l’intérieur de l’espace mémoire réservé à un autre.

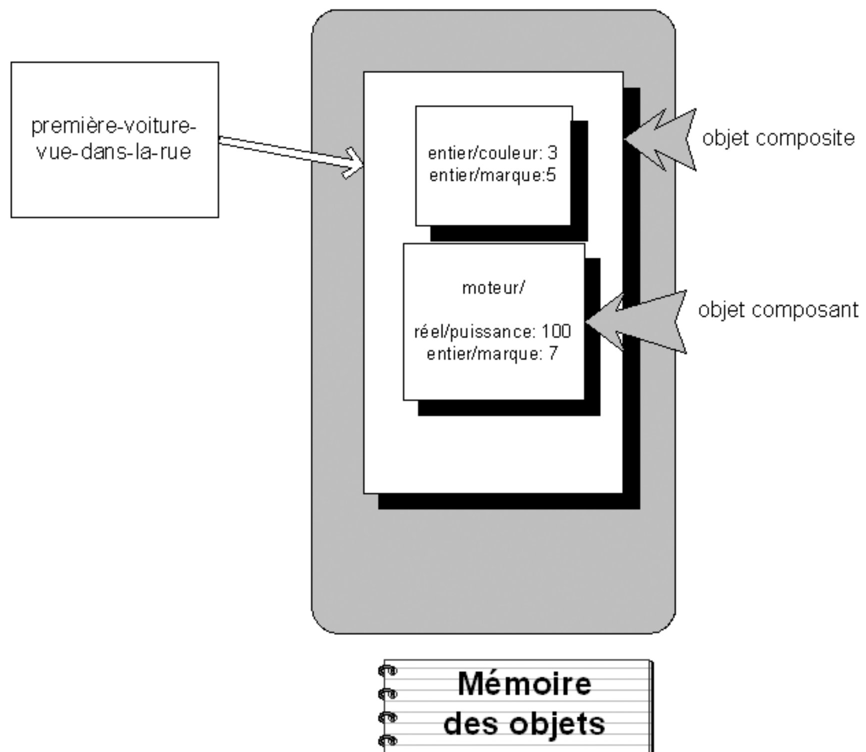


Figure 1-5
L’objet moteur devient un composant de l’objet voiture.

Son accès ne sera dès lors possible qu'à partir de celui qui lui offre cette hospitalité et, s'il le fait, c'est qu'il sait que l'existence de l'hôte est totalement conditionnée par l'existence de l'hôte (la langue française est ainsi faite qu'elle permet cette ambiguïté terminologique !). L'objet moteur, dans ce cas, n'existe que comme seul attribut de l'objet voiture. Si vous vous débarrassez de la voiture, vous vous débarrasserez dans le même temps de son moteur.

Une composition d'objets

Entre eux, les objets peuvent entrer dans une relation de type composition, où certains se trouvent contenus dans d'autres et ne sont accessibles qu'à partir de ces autres. Leur existence dépend entièrement de celle des objets qui les contiennent.

Dépendance sans composition

Vous comprendrez aisément que ce type de relation entre objets ne suffit pas pour permettre une description fidèle de la réalité qui nous entoure. En effet, si la voiture possède bien un moteur, occasionnellement elle contient également des passagers, qui n'aimeraient pas être portés disparus lors de la mise à la casse de la voiture... D'autres modes de mise en relation entre objets devront être considérés, qui permettent à un premier de se connecter facilement à un deuxième, mais sans que l'existence de celui-ci ne soit entièrement conditionnée par l'existence du premier. Mais nous en reparlerons plus tard.

L'objet dans sa version active

Activité des objets

Afin de poursuivre cette petite introspection cognitive dans le monde de l'informatique orientée objet, jetez à nouveau un coup d'œil par la fenêtre et décrivez-nous quelques scènes observées : « une voiture s'arrête à un feu rouge », « les passants traversent la route », « un passant entre dans un magasin », « un oiseau s'envole de l'arbre ». Que dire de toutes ces observations bouleversantes que vous venez d'énoncer ? D'abord, que les objets ne se bornent pas à être statiques. Ils bougent, se déplacent, changent de forme, de couleur, d'humeur, et ce, souvent, suite à une interaction directe avec d'autres objets. La voiture s'arrête car le feu est devenu rouge, et elle redémarre dès qu'il passe au vert. Les passants traversent car les voitures s'arrêtent. L'épicier dit « bonjour » au client qui ouvre la porte de son magasin. Les objets inertes sont par essence bien moins intéressants que ceux qui se modifient constamment. Certains batraciens ne détectent leur nourriture favorite que si elle est en mouvement. Placez-la, immobile, devant eux, et l'animal ne la verra simplement pas. Ainsi, l'objet sera d'autant plus riche d'intérêt qu'il est sujet à des transitions d'états nombreuses et variées.

Les différents états d'un objet

Les objets changent donc d'état, continûment, mais tout en préservant leur identité, en restant ces mêmes objets qu'ils ont toujours été. Les objets sont dynamiques, la valeur de leurs attributs change dans le temps, soit par des mécanismes qui leur sont propres (tel le changement des feux de signalisation), soit en raison d'une interaction avec un autre objet (comme dans le cas de la voiture qui s'arrête au feu rouge).

Du point de vue informatique, rien n'est plus simple que de modifier la valeur d'un attribut. Il suffit de se rendre dans la zone mémoire occupée par cet attribut et de remplacer la valeur qui s'y trouve actuellement stockée par une nouvelle valeur. La mise à jour d'une partie de sa mémoire, par l'exécution d'une instruction appropriée, est une des opérations les plus fréquentes effectuées par un ordinateur. Le changement d'un attribut n'affecte

en rien l’adresse de l’objet, et donc son identité. Tout comme vous, qui restez la même personne, humeur changeante ou non. L’objet, en fait, préservera cette identité jusqu’à sa pure et simple suppression de la mémoire informatique. Pour l’ordinateur : « Partir, c’est mourir tout à fait. ». L’objet naît, vit une succession de changements d’états et finit par disparaître de la mémoire. Et voilà expédié le résumé d’une vie, qu’elle soit digne d’un roman ou d’un simple fait divers. Pas vraiment enviable la vie des objets dans ce monde impitoyable de l’informatique !

> **Changement d’états**
>
> Le cycle de vie d’un objet, lors de l’exécution d’un programme orienté objet, se limite à une succession de changements d’états, jusqu’à sa disparition pure et simple de la mémoire centrale.

Les changements d’état : qui en est la cause ?

Mais qui donc est responsable des changements de valeur des attributs ? Qui a la charge de rendre les objets moins inertes qu’ils n’apparaissent à première vue ? Qui se charge de les faire évoluer et, ce faisant, de les rendre un tant soit peu intéressants ? Reprenons l’exemple des feux de signalisation évoqué plus haut, et comme indiqué à la figure 1-6, stockons-en un, « celui-que-vous-avez-vu-dans-la-rue », dans la mémoire de l’ordinateur (nous supposerons que la couleur est bien représentée par un entier ne prenant que les valeurs 1, 2 ou 3). Installons dans cette même mémoire, mais un peu plus loin, une opération, qui sera responsable du changement de couleur. Dans la mémoire dite centrale, RAM ou vive, d’un ordinateur, ne se trouvent toujours installés que ces deux types d’information, des données et des instructions qui utilisent et modifient ces données, rien d’autre. Nous appellerons la simple opération de changement de couleur : « change ». Comme chaque attribut, chaque opération se doit d’être nommée afin de pouvoir y accéder. Il s’agira pour elle, le plus banalement du monde, d’incrémenter l’entier couleur et de ramener sa valeur à 1 dès que celui-ci atteint 4. C’est cette opération triviale qui mettra un peu d’animation dans notre feu de signalisation.

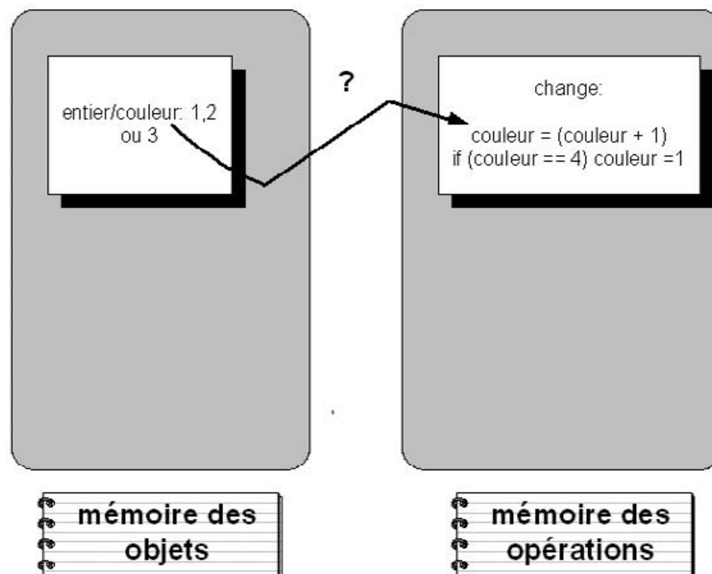


Figure 1-6

Le changement d’état du feu de signalisation par l’entremise de l’opération « change ».

Comment relier les opérations et les attributs ?

Alors que nous comprenons bien l'installation en mémoire, tant de l'objet que de l'opération qui pourra le modifier, ce qui nous apparaît moins évident, c'est la liaison entre les deux. Comment l'opération et les deux instructions de changement (l'incrémentation et le test), installées dans la mémoire à droite, savent-elles qu'elles portent sur le feu de signalisation installé, lui, dans la mémoire à gauche ? Plus concrètement encore, comment l'opération `change`, qui incrémente et teste un entier, sait-elle que cet entier est, de fait, celui qui code la couleur du feu et non « l'âge du capitaine » ? La réponse à cette question vous fait entrer de plain-pied dans le monde de l'orienté objet… Mais vous êtes sans doute un peu ému. Alors, avant de vous donner la réponse tant attendue, permettez-nous de vous souhaiter un chaleureux welcome dans ce monde de l'OO, car vous n'êtes pas près de le quitter.

Introduction à la notion de classe

Méthodes et classes

Place à la réponse. Elle s'articule en deux temps. Dans un premier temps, il faudra que l'opération `change` – que nous appellerons dorénavant *méthode* – ne soit attachée qu'à des objets de type `feu de signalisation`. Seuls ces objets possèdent cet entier à l'endroit où ils le possèdent, et sur lesquels peut s'exercer cette méthode. Appliquer la méthode `change` sur tout autre type d'objet, tel que la voiture ou l'arbre, n'a pas de sens, car on ne saurait de quel entier il s'agit. De surcroît, ce double incrément pour revenir à la valeur de base est totalement dénué de signification pour ces autres objets. Le subterfuge qui permet d'associer, à jamais, les méthodes avec les objets qui leur correspondent consiste à les unir tous deux par les liens, non pas du mariage, mais de la « classe ».

Classe

Une nouvelle structure de données voit le jour en OO : la classe, qui, de fait, a pour principale raison d'être d'unir en son sein tous les attributs de l'objet et toutes les opérations qui y accèdent et qui portent sur ceux-là. Opération que l'on désignera par le nom de *méthode*, et qui regroupe un ensemble d'instructions portant sur les attributs de l'objet. Pour les programmeurs en provenance du procédural, les attributs de la classe sont comme des arguments implicites passés à la méthode ou encore des variables dont la portée d'action se limite à la seule classe.

La classe devient ce contrat logiciel qui lie à vie les attributs de l'objet et les méthodes qui utilisent ces attributs. Par la suite, tout objet devra impérativement respecter ce qui est dit par sa classe, sinon gare au compilateur !

Comme c'est l'usage en informatique s'agissant de variables manipulées, on parlera dorénavant de l'objet comme d'une *instance* de sa classe, et de la classe comme du type de cet objet. Chacun des attributs de l'objet sera « typé » comme il est indiqué dans sa classe, et toutes les méthodes affectant l'objet seront uniquement celles prévues dans sa classe. D'où l'intérêt, bien sûr, de garder une définition de la classe séparée mais partagée par toutes les instances de celles-ci. Non seulement c'est la classe qui déterminera les attributs sur lesquels les méthodes pourront opérer mais, plus encore, et nous accroîtrons dans les prochains chapitres la sévérité de ce principe, seules les méthodes déclarées dans la classe pourront *de facto* manipuler les attributs des objets typés par cette classe. La classe `Feu-de-signalisation` pourrait être définie plus ou moins comme suit :

```
class Feu-de-signalisation {
  int couleur ;
  change() {
    couleur = couleur + 1 ;
    if (couleur ==4) couleur = 1 ;
  }
}
```

> **Définition : type entier**
>
> Le type primitif entier est souvent appelé dans les langages de programmation `int` (pour *integer*), le type réel `double` ou `float`, le caractère `char`. Eh oui ! l'anglais reste l'espéranto de l'informatique.

Chaque objet `feu de signalisation` répondra de sa classe, en faisant en sorte, dès sa naissance, de n'être modifié que par les méthodes déclarées dans sa classe (ici la seule méthode `change`, mais il pourrait y en avoir bien d'autres comme `met-le-feu-en-stand-by`, `change-la-durée-d'une-des-couleurs`, et il faudrait alors rajouter quelques attributs comme la durée de chaque couleur). Gardez bien à l'esprit ce principe fondateur de l'OO qu'aucune autre méthode, jamais, que celles prévues par la classe, ne pourra s'aventurer à changer la valeur des attributs des objets de cette classe. Ce qui permet aux objets de se modifier leur est aussi propre que les attributs qui les décrivent structurellement. Un objet existe, par l'entremise de ses attributs, et se modifie, par l'entremise de ses méthodes (et nous disons bien « ses » et non « ces » ou vous risquez d'être bouté à jamais hors du royaume merveilleux de l'OO).

Sur quel objet précis s'exécute la méthode

Dans un second temps, il faudra signaler à la méthode `change` (qui maintenant, grâce à la définition de la classe, sait qu'elle n'opère exclusivement que sur des feux de signalisation) lequel, parmi tous les feux possibles et stockés en mémoire, est celui qu'il est nécessaire de changer. Cela se fera par le simple appel de la méthode sur l'objet en question, et, plus encore, par l'écriture d'une instruction de programmation de type :

```
feu-de-signalisation-en-question.change()
```

Nous appliquons la méthode `change()` (nous expliquerons plus tard la raison d'être des parenthèses) sur l'objet `feu-de-signalisation-en-question`. C'est le point dans cette instruction qui permet ici la liaison entre l'objet précis et la méthode à exécuter sur cet objet. N'oubliez pas que le référent `feu-de-signalisation-en-question` possède effectivement l'adresse de l'objet et, de là, automatiquement, de l'attribut entier/couleur sur lequel la méthode `change()` doit s'appliquer.

> **Lier la méthode à l'objet**
>
> On lie la méthode `f(x)` à l'objet « a », sur lequel elle doit s'appliquer, au moyen d'une instruction comme : `a.f(x)`. Par cette écriture, la méthode `f(x)` saura comment accéder aux seuls attributs de l'objet, ici les attributs de l'objet « a », qu'elle peut manipuler.

> **Différencier langage orienté objet et langage manipulant des objets**
>
> De nombreux langages de programmation, surtout de scripts pour le développement web (JavaScript, VB Script), rendent possible l'exécution de méthodes sur des objets dont les classes préexistent au développement. Le programmeur ne crée jamais de nouvelles classes mais se contente d'exécuter les méthodes de celles-ci sur des objets. Supposons par exemple que vous vouliez agrandir un « font » particulier de votre page web. Vous écrirez `f.setSize(16)` mais jamais dans votre code, vous n'aurez créé la classe `Font` (vous utilisez l'objet « f » issu de cette classe) et sa méthode `setSize()`. Vous vous limitez à les utiliser comme vous utilisez les bibliothèques d'un quelconque langage de programmation. Les classes inclues dans ces librairies auront été développées par d'autres programmeurs et mis à votre disposition.

Voilà, c'est aussi simple que cela, et c'est le départ d'un grand voyage dans le monde de l'OO, un voyage qui nous réserve encore de nombreuses surprises.

Des objets en interaction

Parmi les saynètes évoquées plus haut, certaines décrivaient une interaction entre deux objets, comme le feu qui, passant au vert, permet à la voiture de démarrer, ou l'épicier qui salue le nouveau client. Ce serait bien qu'à l'instar de cette réalité observée, les objets informatiques puissent interagir de la sorte. Vous allez être contents. C'est non seulement possible, mais c'est la base de l'informatique OO. Rien d'autre, en effet, ne se passe dans cette informatique-là que des objets interagissant entre eux. Dans le monde, un objet esseulé n'est pas grand-chose, en OO également. C'est ensemble, mais aussi et paradoxalement chacun pour soi (ce paradoxe sera résolu plus tard), qu'ils commencent à nous être utiles. Tentons d'imaginer comment la première scène, décrivant l'effet du changement de couleur du feu sur le démarrage de la voiture, pourrait être reproduite informatiquement. Nous considérerons que les deux objets, `feu-de-signalisation` et `voiture-vue-dans-la-rue`, instance pour le premier d'une classe `Feu-de-signalisation` (notez la majuscule de la première lettre) et pour le deuxième d'une classe `Voiture`, sont chacun caractérisés par un attribut, l'entier couleur pour le feu, et, pour la voiture, un entier vitesse pouvant prendre jusqu'à 130 valeurs possibles (en tout cas en France, le traducteur allemand s'adaptera).

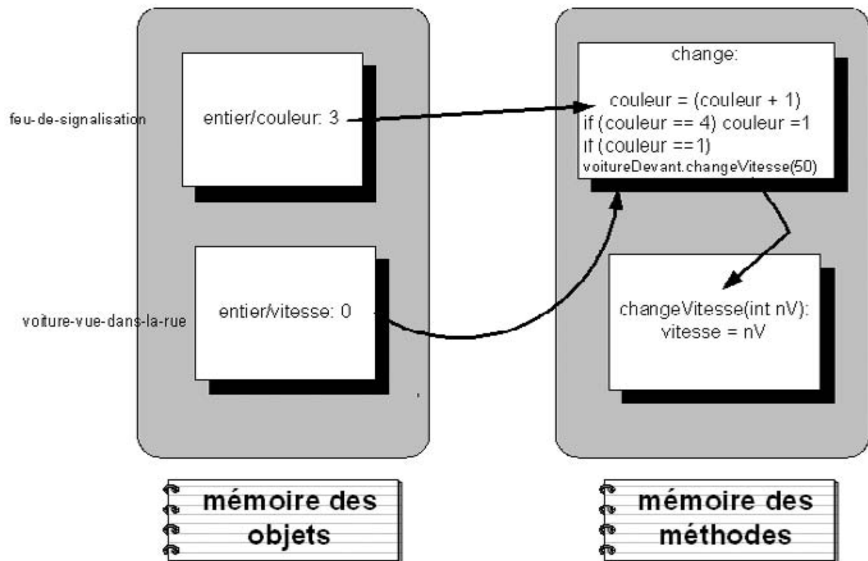


Figure 1-7
Comment l'objet « feu-de-signalisation » parle-t-il à l'objet « voiture-vue-dans-la-rue » ?

Comment les objets communiquent

Faisons simple, quitte à faire irréaliste, en supposant que le changement de couleur du feu induise dans la voiture l'accélération de la vitesse de 0 à 50. Observez la figure 1-7, comme pour la couleur du feu, dont les seules modifications ne sont permises que par la méthode `change` ; le changement de vitesse de la voiture relèvera, également, exclusivement de l'exécution d'une méthode, que nous dénommerons `changeVitesse(int nV)` (car d'autres attributs de la voiture pourraient également faire l'objet de changement). Nous constatons, par ailleurs, qu'il est prévu que cette méthode reçoive un argument de type entier, qui lui permette d'affiner son effet en fonction de la valeur de cet argument.

Les plus informaticiens d'entre vous noteront que l'écriture, la syntaxe et le mode de fonctionnement d'une méthode sont en tous points semblables aux routines ou procédures dans un quelconque langage de programmation. Elles peuvent recevoir des arguments, qu'elles utiliseront dans le corps de leur définition, et ce afin de paramétrer leur fonctionnement, comme ici pour la méthode `changeVitesse(int nV)`. de la classe `Voiture`. C'est la raison d'être des parenthèses qui, même lorsqu'elles ne contiennent aucun argument, sont obligées d'apparaître dans l'appel de la méthode.

Méthode

Il s'agit d'un regroupement d'instructions semblable aux procédures, fonctions et routines rencontrés dans tous les langages de programmation, à ceci près qu'une méthode s'exécute toujours sur un objet précis (comme si celui-ci lui était, implicitement, passé comme un argument additionnel).

Envoi de messages

D'ores et déjà, vous en aurez déduit que la seule manière pour deux objets de communiquer, c'est que l'un demande à l'autre d'exécuter une méthode qui lui est propre. Ici, le feu de signalisation demande à la voiture d'exécuter sa méthode `changeVitesse(50)`, qui lui permet de modifier son attribut vitesse et de le porter à 50. Rappelez-vous qu'il serait impropre que le feu s'en charge directement, étant donné que seules les méthodes de la classe `Voiture` peuvent se permettre de modifier l'état de cette dernière. En se référant à la figure 1-7, vous constatez que le moyen utilisé par l'objet `feu` pour déclencher la méthode `changeVitessse` est de prévoir dans le corps de sa propre méthode une instruction telle que `voitureDevant.changeVitesse(50)`. Le feu s'adresse donc à un référent particulier dénommé `voitureDevant`, sur lequel il déclenche la méthode `changeVitesse`.

Envoi de message

Le seul mode de communication entre deux objets revient à la possibilité pour le premier de déclencher une méthode sur le second, méthode déclarée et définie dans la classe de celui-ci. On appellera ce mécanisme de communication un « envoi de message » du premier objet vers le second. Cette expression se justifie par la présence de ces deux objets : l'expéditeur dans le code duquel l'envoi se produit et le destinataire à qui le message est destiné. Elle se justifie davantage encore pour des objets s'exécutant sur des ordinateurs très éloignés géographiquement, situation entraînant réellement un envoi physique de message d'un point à l'autre du globe.

Identification des destinataires de message

On comprend bien la démarche de l'objet feu, mais tout informaticien restera quelque peu interloqué par la brutalité d'exécution. Ça marche cette recette ? Non, bien sûr ! Il nous manque quelques ingrédients indispensables. Le premier est de signaler au feu que ce référent `voitureDevant` est bien de type classe `Voiture` et que, de ce fait, cette demande d'exécution de méthode est tout à fait légitime. Afin de typer ce référent-là, on pourrait tout simplement le passer comme argument de la méthode `change` pour le feu. Cependant, ce que nous rencontrerons bien plus souvent, c'est le procédé qui consiste à faire de ce référent un attribut à part entière de la classe `Feu-de-signalisation`, un attribut de type `Voiture`. La classe `Feu-de-signalisation` se définirait alors comme ceci :

```
class Feu-de-signalisation {
  int couleur ;
  Voiture voitureDevant;
```

```
    change() {
      couleur = couleur + 1 ;
      if (couleur ==4) couleur = 1 ;
      if (couleur ==1) voitureDevant.changeVitesse(50) ;
    }
  }
```

Plus rien n'est vraiment choquant dans cette écriture, car le référent `voitureDevant` étant en effet de type classe `Voiture`, il peut recevoir le message `changeVitesse(50)`. Le compilateur prendra garde de vérifier qu'il existe bel et bien à proximité une classe Voiture contenant la méthode `changeVitesse(int)`. Nous reviendrons sur ce mécanisme dans les prochains chapitres. Syntaxiquement, cette écriture est parfaitement correcte, y compris en l'absence pour l'instant du moindre objet voiture. Ce qui est simplement dit à ce stade de l'écriture de la classe, est que tout objet feu de signalisation se voit associer un objet voiture auquel il peut envoyer le message `changeVitesse(x)`. Une association est ici réalisée entre les classes `Feu-de-signalisation` et `Voiture` et elle va dans le sens du `Feu` vers la `Voiture`. Nous introduirons les diagrammes de classe UML plus avant dans le livre, mais d'ores et déjà cette association dirigée entre la classe `Feu-de-signalisation` et la classe `Voiture` se représentera comme suit en UML.

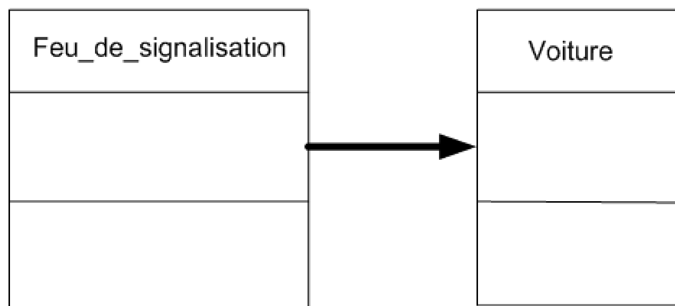


Figure 1-8
Un premier diagramme UML.

Le deuxième ingrédient indispensable est de relier ce référent à l'objet en question, celui que nous désirons voir démarrer quand le feu passe au vert, et donc d'assigner à ce référent la même adresse physique que celle contenue dans le référent `voiture-vue-dans-la-rue`. Nous décrirons par la suite différentes manières d'y parvenir. Mais si nous adoptons l'écriture de la classe indiquée plus haut, une simple manière consistera à prévoir, au cours de la création de l'objet `feu-de-signalisation`, une instruction telle que : `voitureDevant = voiture-vue-dans-la-rue`, qui permettra à l'adresse physique de la voiture en question d'être directement transmise au feu-de-signalisation. Dorénavant l'attribut de la classe `Feu` ayant pour mission de référer l'objet `voiture` avec lequel l'objet `feu` se doit d'interagir possédera en effet l'adresse mémoire de celui-ci.

Gestion d'événement

Lorsqu'il passe au vert, plutôt qu'un envoi de message du feu destiné à toutes voitures qui lui font face (et dont il ignore la nature et le nombre), il serait sans doute plus réaliste de considérer que les voitures sont susceptibles d'observer la transition de couleur du feu et de réagir en conséquence, c'est-à-dire démarrer, sans en être explicitement « ordonné » par le feu. Ce mécanisme d'observation et de gestion d'événement est également un « plus » de la programmation OO et le chapitre 18, qui y est consacré, vous indiquera comment, en effet, les voitures pourraient réagir au quart de tour au changement de couleur, sans recevoir le moindre message explicite du feu.

Des objets soumis à une hiérarchie

Dernier petit détour du côté de la fenêtre (dernier, promis !) et, rassurez-vous, vous n'aurez plus à regarder quoi que ce soit, mais plutôt à vous interroger sur la manière dont, tout à l'heure, vous avez nommé les objets.

Du plus général au plus spécifique

Vous avez parlé de « voiture », « passant », « arbre », « immeuble ». Prenons le premier de ces objets. Vous avez dit « voiture » mais êtes-vous vraiment sûr qu'il s'agissait d'une voiture ? Ne s'agissait-il pas plutôt, pour être précis, d'une Peugeot, plus encore d'une 206, ou, de surcroît, dans sa version turbo ou cabriolet. Qu'est-ce que cela peut bien changer, direz-vous ? Pas grand-chose ici, mais beaucoup pour l'informatique OO, car ce seul et même objet, celui que vous avez vu, est, en fait, tout cela à la fois. Ainsi, pour être tout à fait complet, il aurait également pu être qualifié de « moyen de transport » ou « juste un objet de ce monde ». Il l'est d'ailleurs. Il est bien ces six ou sept concepts, tout à la fois. Tous ces différents concepts existent et forment entre eux une hiérarchie ou taxonomie : du plus général au plus spécifique. Et c'est ce qui nous permet de les utiliser de la manière la plus adaptée et la plus économique qui soit.

Héritage et taxonomie

Une pratique clé de l'orienté objet est d'organiser les classes entre elles de manière hiérarchique ou taxonomique, des plus générales aux plus spécifiques. On parlera d'un mécanisme « d'héritage » entre les classes. Un objet, instance d'une classe, sera à la fois instance de cette classe mais également de toutes celles qui la généralisent et dont elle hérite. Tout autre objet ayant besoin de ses services choisira de le traiter selon le niveau hiérarchique le plus approprié. Pour vous lecteurs, nous ne sommes que de pauvres objets enseignants de la chose informatique. Si vous nous connaissiez mieux, vous découvririez des natures autrement plus raffinées, mais à quoi cela vous servirait-il de mieux nous connaître ?

Vous allez être surpris en apprenant que, tout bien pensé, il n'existe dans ce monde aucune voiture, tout comme il n'existe aucun arbre. D'ailleurs, nous expliquerons plus tard pourquoi, malgré l'existence possible de classes `Arbre` ou `Voiture` dans le logiciel OO que nous pourrions réaliser, il serait souhaitable que ces classes ne donnent naissance à aucun objet. En revanche, il existe des Peugeot 206, des Renault Kangoo, des Fiat Uno, des Volkswagen Golf. Il existe des peupliers, des cerisiers, et même des cerisiers du Japon. Pourquoi alors ce concept de voiture, si rien de ce que nous percevons ne s'y rapporte vraiment ?

C'est parce que l'usage que l'immense majorité des êtres humains font de leur objet voiture et les événements les plus fréquents qu'ils narrent, liés à ce même objet, ne requièrent aucunement d'en connaître la marque : « J'ai pris la voiture pour partir en voyage », « Ma voiture est en panne », « J'ai eu un accident de voiture ». Ce serait la même histoire, le même scénario, si vous remplaciez la voiture par sa marque. Dès lors, cette précision devient inutile car elle n'apporte rien de plus à la conversation, et risque même de détourner le sens premier de vos propos. En outre, le même traitement est souvent réservé à toutes les voitures, quelle que soit leur marque. Il est bien commode de pouvoir dire, dans une seule et même phrase : « Les voitures font la queue devant la station service », « L'accident a impliqué cinq voitures », « Après deux ans, votre voiture doit passer au contrôle technique ».

Dépendance contextuelle du bon niveau taxonomique

Dans l'emploi que vous faites du concept « voiture », lors de conversations, rêveries, écritures, ce simple mot « voiture » suffit largement à véhiculer tout le sens qui est nécessaire à ces contextes. De même, généraliser

d'un cran ce concept, et parler de « moyen de transport » en lieu et place de « voiture », risque, là encore, de dénaturer le sens de vos propos. Car ce niveau intègre également des objets comme le train et l'avion, et votre interlocuteur ne pourra manquer de généraliser vos propos à ces autres objets. Le niveau taxonomique que vous utilisez dépend bien évidemment du contexte. Dialoguant avec un garagiste, il y a fort à parier que vous serez contraint à un moment ou à un autre de lui préciser la marque de la voiture, mais cela se produira rarement dans la grande majorité des interactions sociales. Croyez-nous ou croyez Wittgenstein (c'est plus sûr), qui est une des figures intellectuelles les plus marquantes de ce siècle : c'est l'utilisation que vous en faites, plus que la réalité qu'ils dépeignent, qui sous-tend le sens des mots. Les mots sont d'abord des outils au service de nos interactions sociales ou de nos élucubrations mentales. Comme dans la majorité de celles-ci, le mot « voiture » suffit, non seulement à vous véhiculer, mais également à véhiculer tout ce qu'il vous est important de signifier à son propos, d'abord à vous puis aux autres, c'est pour cela que vous nous avez parlé de voiture, tout à l'heure, en regardant par la fenêtre.

Wittgenstein

Cette figure de légende de la philosophie, né à Vienne en 1889 et mort à Cambridge en 1951, a vécu mille vies, toutes plus extraordinaires les unes que les autres, et bâtit deux ouvrages philosophiques parmi les plus marquants et illustres du XXe siècle. Il est issu d'une des familles les plus riches d'Autriche, cadet d'une famille de huit enfants, tous marqués par un destin cruel. Jeune et brillantissime, il se destine à une carrière d'ingénieur aéronautique prometteuse, mais finit par s'en détourner pour, au contact des mathématiciens et des philosophes de Cambridge, tel Russel, se lancer dans sa première œuvre philosophique, consacrée à la nature du langage et de la pensée : le *Tractacus*. Dans cette œuvre, il accorde au langage des vertus figuratives, le disséquant pour le présenter en une composition d'objets atomiques, isomorphes au monde, et rentrant dans des relations structurelles, tout aussi fidèle à la réalité qu'il cherche à dépeindre. Ce premier Wittgenstein est un précurseur de nos objets informatiques et des liens relationnels qu'ils maintiennent entre eux.

Ensuite, héros de la Première Guerre mondiale (il s'adonne à ses écrits philosophiques entre deux obus), maintenant avec le monde universitaire un rapport haine/amour, des allées et venues incessantes, assistant à Cambridge, instituteur dans d'austères villages de montagne, jardinier de couvent, architecte pour la maison de sa sœur, il se décide à remettre complètement en question la vision du langage présentée dans ses premiers écrits, et qui, pourtant, fait autorité dans son cercle universitaire. C'est le deuxième Wittgenstein, auteur des *Recherches philosophiques*, et qui retire dorénavant au langage ces mêmes vertus figuratives dont il l'a paré dans une première vie, pour, à la place, l'envisager comme un outil d'interaction sociale, dont la quintessence sémantique est à puiser dans son usage plutôt que dans la réalité qu'il dépeint. Ce deuxième Wittgenstein nous permet de comprendre l'intérêt des mécanismes d'héritage, qui est à trouver davantage dans la manière et les contextes d'utilisation de nos mots que dans les objets du réel que ces mots désignent. Il n'y a, de fait, ni voiture ni arbre dans le monde, pas plus du temps de Wittgenstein qu'aujourd'hui, mais il nous est bien utile de pouvoir recourir à ces mots dans tant de situations.

Il continuera cette vie dissolue, entre le monde académique, les salles d'hôpitaux où il est homme à tout faire, pour terminer sa vie dans une hutte de pêcheur, où il commence à s'éteindre, se débattant dans la maladie, la solitude et la tourmente mentale. Entre-temps, cet incontestable génie se sera débarrassé au profit d'artistes en peine et de plus pauvres de l'immense fortune héritée de son père. Il passera l'essentiel de ses loisirs au cinéma, à voir des polars et des westerns qu'il privilégie à toute autre stimulation mentale. Il vivra très difficilement son homosexualité. Étrange destin décidément que celui de Wittgenstein qui, partageant, petit, les bancs d'école avec un certain Adolphe, aurait non seulement été (selon certains) à l'origine de la haine que son camarade d'école voua au peuple juif, mais fut un héros de la résistance et de l'espionnage britannique, pendant la Seconde Guerre mondiale. Cela vaut bien ce petit encart, non ?

> **Héritage**
>
> Dans notre cognition et dans nos ordinateurs, le rôle premier de l’héritage est de favoriser une économie de représentation et de traitement. La factorisation de ce qui est commun à plusieurs sous-classes dans une même superclasse offre des avantages capitaux. Vous pouvez omettre d’écrire dans la définition de toutes les sous-classes ce qu’elles héritent des superclasses. Il est de bon sens que, moins on écrit d’instructions, plus fiable et plus facile à maintenir sera le code. Si vous apprenez d’une classe quelconque qu’elle est un cas particulier d’une classe générale, vous pouvez lui associer automatiquement toutes les informations caractérisant la classe plus générale et ce, sans les redéfinir. De plus, vous ne recourrez à cette classe plus spécifique que dans des cas bien plus rares, où il vous sera essentiel d’exploiter les informations qui lui sont propres.

Polymorphisme

Plusieurs voitures patientent, le moteur ronronnant et l’automobiliste la bave aux lèvres, devant le feu. Dès que le feu passe au vert, c’est avec rage que tous ces moteurs s’emballent et propulsent leur voiture. Elles démarrent toutes, de fait, mais pas de la même manière. La pauvre 2-CV, après quelques soubresauts et quelques protestations mécaniques, cale péniblement. La Twingo s’avance tranquillement dans le carrefour, le temps pour son conducteur de sourire par la fenêtre au malheureux conducteur de la 2-CV. La BMW la double férocement et traverse le carrefour en moins de temps qu’il ne faut pour le dire. En réalité, toutes ces voitures ont bien reçu le même message de démarrage, envoyé par le feu, mais se sont empressées de l’interpréter différemment. Et c’est tant mieux pour le feu, qui serait bien en peine de différencier le message en fonction des voitures à qui il les adresse.

Notre conceptualisation du monde, par héritage et généralisation, est ainsi faite, que nous retrouvons la même dénomination pour des activités partagées par un ensemble d’objets, mais dont l’exécution se particularise en fonction de la vraie nature de ces objets. Cela permet à un premier objet, interagissant avec cet ensemble d’objets, dont il sait qu’ils sont à même d’exécuter ce message, de le leur adresser sans se préoccuper de cette nature intime. L’objet feu n’a que faire dans son fonctionnement de la marque des voitures avec lesquelles il communique. Pour lui, il s’agit là uniquement d’objets de la classe voiture, objets qui peuvent tous démarrer, un point c’est tout. Une grande économie de conception et un gage de stabilité sont permis par ce mécanisme (ajouter une nouvelle sous-classe de voiture devant le feu ne changera rien au comportement de ce dernier), dont le nom vous permettra de briller dans les salons : *polymorphisme*.

Prenez la souris de votre PC, cliquez partout sur votre écran et regardez ce qui se passe : des menus se déroulent, des fenêtres s’ouvrent, d’autres se ferment, des icônes s’inscrivent, des petits « Einstein » vous cassent les pieds. Pourtant, tous les objets de votre écran reçoivent ce même clic, mais tous l’interprètent différemment. Un seul clic et autant de réaction à celui-là qu’il n’y a de types d’objets sur votre écran. Vous, objets lecteurs et apprentis informaticiens, nous, auteurs, nous vous incitons à lire ce livre, en prévoyant que vous le lirez, tous à votre rythme, et en l’appréciant différemment, suivant vos prérequis, votre enthousiasme à la lecture et votre goût pour l’informatique. Y compris ceux qui le ferment à l’instant même et le jettent au bout du lit, vous êtes polymorphes et soyez-en fiers !

> **Polymorphisme, conséquence directe de l’héritage**
>
> Le polymorphisme, conséquence directe de l’héritage, permet à un même message, dont l’existence est prévue dans une superclasse, de s’exécuter différemment, selon que l’objet qui le reçoit est d’une sous-classe ou d’une autre. Cela permet à l’objet responsable de l’envoi du message de ne pas avoir à se préoccuper dans son code de la nature ultime de l’objet qui le reçoit et donc de la façon dont il l’exécutera.

Héritage bien reçu

Et c'est avec ce mécanisme d'héritage que nous terminons notre entrée dans le monde de l'OO, muni, comme vous vous en serez rendu compte, de notre petit manuel de psychologie. Rien de bien surprenant à cela. Les sciences cognitives et l'intelligence artificielle prennent une large place dans le faire-part de naissance de l'informatique OO. Dans les sciences cognitives, cette idée d'objet est largement répandue, déguisée sous les traits des schémas piagétiens, des noumènes kantiens (et en avant pour la confiture…), des paradigmes kuhniens. Tous ces auteurs se sont efforcés de nous rappeler que notre connaissance n'est pas aussi désorganisée qu'elle n'y paraît. Que des blocs apparaissent, faisant de notre cognition un cheptel d'îles plutôt qu'un océan uniforme, blocs reliés entre eux de manière relationnelle et taxonomique. Cette structuration cognitive reflète, en partie, la réalité qui nous entoure, mais surtout notre manière de la percevoir et de la communiquer, tout en se soumettant à des principes universaux d'économie, de simplicité et de stabilité.

Exercices

Exercice 1.1

Prenez comme exemple une de vos activités sportives, culturelles, artistiques ou sociales, et faites une liste des objets impliqués dans cette activité. Dans un premier temps, créez ces objets sous formes de couples attribut/valeur. Dans un deuxième temps, réfléchissez au lien d'interaction existant entre ces objets ainsi qu'à la manière dont ils sont capables de s'influencer mutuellement. Dans un troisième temps, identifiez pour chaque objet une possible classe le caractérisant.

Exercice 1.2

Répondez aux questions suivantes :

- un même référent peut-il désigner plusieurs objets ?
- plusieurs référents peuvent-ils désigner un même et seul objet ?
- un objet peut-il faire référence à un autre ? si oui, comment ?
- pourquoi l'objet a-t-il besoin d'une classe pour exister ?
- un objet peut-il changer d'état ? si oui, comment ?
- que signifie cette écriture `a.f(x)` ?
- où doit être déclarée `f(x)` pour que l'instruction précédente s'exécute sans problème ?
- qu'appelle-t-on un envoi de message ?
- comment un premier objet peut-il agir de telle sorte qu'un deuxième objet change d'état suite à cette action ?

Exercice 1.3

Placez dans un arbre taxonomique, du plus général au plus spécifique, les concepts suivants :

- humain, footballeur, avant-centre, sportif, skieur, spécialiste du slalom géant ;
- guitare, instrument de musique, trompette, instrument à vent, instrument à corde, violon, saxophone, voix.

Exercice 1.4

Réfléchissez à quelques objets de votre entourage : livre, ordinateur, portefeuille, collègues, téléphone, et interrogez-vous à chaque fois sur le niveau taxonomique que vous privilégiez dans la manière de les désigner. Pourquoi celui-là ? Par exemple, pourquoi dites-vous simplement livre et pourquoi pas le livre d'Eyrolles intitulé « L'orienté objet » ?

Exercice 1.5

Dans les couples d'objets suivants : voiture/chauffeur, footballeur/ballon, guitare/guitariste, télévision/ télécommande, lequel des deux est l'expéditeur et lequel est le destinataire des messages ?

2

Un objet sans classe… n’a pas de classe

Ce chapitre a pour but d’introduire la notion de classe : de quoi une classe est-elle faite et quel rôle joue-t-elle, durant le développement du programme, sa compilation, sa structuration finale, et surtout son découpage. On verra que les classes servent à la fois de modèle à respecter stricto sensu *par les objets, ainsi que de modules idéaux pour l’organisation logicielle.*

Sommaire : Classe — Méthode — Signature de méthode — La classe, vigile de son bon usage — Méthodes et attributs statiques — La découpe logicielle en classe

Candidus — Enfin, vas-tu m’expliquer le mode d’emploi de ton bébé, oui ou non ?

Doctus — La classe…

Cand. — Eh bien, il grandit vite !

Doc. — Bon ! Allons-y à fond dans le genre imagé. Le mode procédural consistait à chatouiller bébé au bon endroit pour le forcer à faire ce qu’on attendait de lui, le mode orienté objet consiste maintenant à lui fournir des moyens d’agir qui lui sont accessibles. Si nous nous arrangeons pour organiser son environnement comme un ensemble de modules simples et complets, il nous fera tout un tas de petits miracles.

Cand. — Et on sait bien que beaucoup de programmes marchent par miracle.

Doc. —… Je continue. Ces modules ne seront pas autre chose qu’un ensemble de pièces avec leurs règles d’utilisation.

Cand. — J’imagine que tous ces petits modules sont en fait les différents composants d’une structure. Ce n’est que la vision globale de l’ensemble qui laissera voir la complexité du travail de bébé. Ça semble génial… Tu viens de faire la même découverte que Descartes quand il voulait tout expliquer en réduisant tout ce qui lui apparaissait complexe en des parties plus simples !

Doc. — Je ne prétends pas tout expliquer! Je parle d’une simple orientation de l’effort à fournir. C’est toi qui devras tout expliquer quand il te faudra réaliser un programme particulier. Ce qu’il faut retenir de cette orientation est que ton effort devra être basculé de la phase de développement vers la phase de conception. Tes données ne seront plus ces choses inertes avec lesquelles tu jonglais en te servant de fonctions bien trop complexes, ce seront des acteurs à part entière de ton programme. De leur plein gré, elles sauront quoi faire et avec qui le faire.

Cand. — Là, tu commences à m’intéresser !

Doc. — Ah ! parce que ce n’est qu’à ce chapitre que tu trouves ça intéressant !

Constitution d'une classe d'objets

Le premier chapitre a apporté une première justification à la nécessité de faire précéder toute manipulation d'objets d'une structure de données, associant aux attributs de l'objet les seules méthodes qui peuvent y avoir accès. Dorénavant, chaque objet créé le sera à partir d'une classe à laquelle il sera tenu de se conformer tout au long de son existence. Rien n'interdit pourtant dans la réalité, un objet de changer de statut ou de comportement, par exemple un étudiant de devenir professeur ou un professeur d'informatique de devenir sommelier. C'est possible dans la réalité mais pas dans l'OO d'aujourd'hui, peut-être de demain, l'objet est coincé par sa classe, même s'il s'y sent parfois à l'étroit. Dans les langages OO, la classe est le modèle à respecter *stricto sensu*, comme une maison le fait du plan de l'architecte, et la robe du patron du couturier. La classe se décrit au moyen de trois informations, (voir figure 2-1).

Programmation orientée *objet*, ou programmation orientée *classe* ?

En réalité, un programmeur OO passe bien plus de temps à *faire ses classes* qu'à se préoccuper de ce qu'il adviendra aux objets quand ces derniers s'exécuteront en effet. Plutôt que « programmation orientée objet », il aurait été plus adéquat de qualifier ce type de programmation d'« orientée classe » !

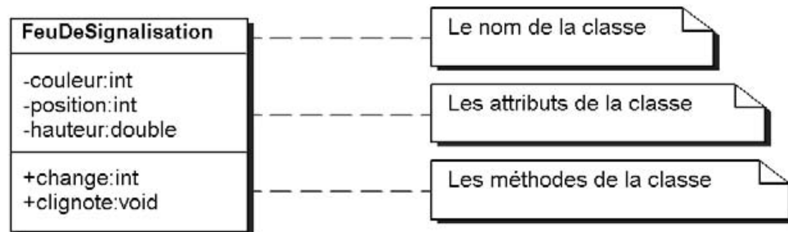


Figure 2-1
Un exemple d'une classe et des trois types d'information qui la composent.

Les trois informations constitutives de la classe

Toute définition de classe inclut trois informations : d'abord, le nom de la classe, ensuite ses attributs et leur type, enfin ses méthodes.

Le nom de la classe, ici : `FeuDeSignalisation`, le nom des attributs : `couleur`, `position` et `hauteur`, et leur type : `int` , `int` (il s'agit de deux entiers) et `double` (il s'agit d'un réel), enfin, le nom des méthodes `change`, `clignote`, avec la liste des arguments et le type de ce que les méthodes retournent.

Définition d'une méthode de la classe : avec ou sans retour

Une méthode retourne quelque chose si le corps de ses instructions se termine par une expression telle que « return x ». Si c'est le cas, son nom sera précédé du type de ce qu'elle retourne. Par exemple, la méthode `change`, modifiant la couleur du feu, pourrait se définir comme suit :

```
int change() {
  couleur = couleur + 1 ;
  if couleur == 4 couleur = 1;
    return couleur ; /* la méthode retourne un entier */
}
```

Commentaires

`/* …. */` encadre des commentaires à l'intérieur d'un code. Lorsque les commentaires restent sur une seule ligne, on peut également utiliser `//`. Toute écriture mise en commentaire est désactivée dans le code. Nous utiliserons beaucoup les commentaires dans nos codes, de manière à expliquer ceux-ci sans pour autant modifier la façon dont ils s'exécutent. Pour Python, en revanche, tous les commentaires débutent par le dièse #.

La couleur étant représentée par un entier, le retour de la méthode est de type entier. La rencontre du mot `return` met fin à l'exécution de la méthode en replaçant celle-ci dans le code qui l'appelle par la valeur de ce retour. La différence entre une méthode qui retourne quelque chose et une méthode qui ne retourne rien (`void` précède alors le nom de la méthode) se marque uniquement dans le contexte d'exécution de la méthode.

La seconde méthode de la classe, `clignote()`, ne retourne rien. Son appel dans un corps d'instruction se fera indépendamment d'un contexte opératoire spécifique, alors que l'appel de la méthode `change()` pourra (car elle pourrait être également appelée comme une méthode qui ne retourne rien) se produire à l'intérieur d'une expression. Dans cette expression, l'appel de cette méthode, dans son entièreté, pourrait être remplacé par un simple entier, comme dans :

```
if  (change() == 1)  print (« le feu est vert »)
```

ou encore *:*

```
int b = change() + 2 ;
```

Fonctions et procédures

Les praticiens des langages de programmation procéduraux retrouveront là la distinction faite généralement dans ces langages entre une fonction (déclarée avec retour comme toute fonction mathématique f(x) en général) et une procédure (déclarée sans retour et qui se borne à modifier des données du code sans que cette action soit intégrée à l'intérieur même d'une instruction).

De même, une méthode, comme toute opération informatique (fonction ou procédure), peut recevoir un ensemble *d'arguments* entre les parenthèses, qu'elle utilisera dans le cours de son exécution. Dans l'exemple ci-après, l'argument entier « a » permet de calibrer la boucle présente dans la méthode. Le corps de cette méthode fait clignoter le feu deux fois et peut, en fonction de la valeur de « a », adapter la durée des phases éteintes et allumées.

```
void clignote(int a) {
  System.out.println("deuxieme maniere de clignoter"); /* Affichage de texte à l'écran */
  for(int i=0; i<2; i++) {
    for (int j=0; j<a; j++)  // on retrouve le "a" de l'argument
      System.out.println("je suis eteint");
    for (int j=0; j<a; j++)
      System.out.println("je suis allume");
  }
}
```

Arguments de méthode

La présence des arguments dans la définition d'une méthode permet de moduler le comportement du corps d'instructions de cette méthode selon la valeur prise par ces arguments. C'est l'équivalent du x dans les fonctions mathématiques f(x).

Identification et surcharge des méthodes par leur signature

Ensemble, le nom de la méthode ainsi que la liste et la nature de ses arguments, constituent la *signature* de cette méthode. Tout envoi de message est conditionné par cette signature. Si un objet parle à un autre, le langage d'interaction sera la liste des signatures des méthodes disponibles chez cet autre. Cette signature est associée de manière unique au corps d'instructions qui composent la méthode et qui, *in fine*, l'exécuteront. Cette signature est à rapprocher du référent des objets, car il s'agit à nouveau d'un mode d'accès. Cette signature peut faire référer à un autre corps d'instructions dès lors que, tout en conservant son nom, elle modifie quoi que ce soit dans la liste ou dans le type de ses arguments. Si pour un même nom, on modifie dans une nouvelle définition de méthode la seule liste des arguments, on parle alors de surcharge de méthodes. Une modification dans cette liste des arguments (changement du nombre ou du typage de ceux-ci) donnera lieu à une nouvelle méthode, car lors de l'appel de la méthode, le choix de la version dépendra du nombre et du type des arguments.

Surcharge de méthode

La manœuvre consistant à surcharger une méthode revient à en créer une nouvelle, dont la signature se différencie de la précédente, uniquement par la liste ou la nature des arguments.

Il n'est pas possible d'avoir deux méthodes qui possèdent la même signature, c'est-à-dire le même nom et la même liste d'arguments, et qui se différencient par le contexte opérationnel dans lequel elles sont appelées, comme le type de « retour ». Et ce parce que, au moment précis de l'appel d'une des deux méthodes, elles resteront indistinguables.

Signature de méthode

La signature de la méthode est ce qui permet de la retrouver dans la mémoire des méthodes. Elle est constituée du nom, de la liste, ainsi que du type des arguments. Toute modification de cette liste pourra donner naissance à une nouvelle méthode, surcharge de la précédente. La nature du `return` ne fait pas partie de cette signature dans la mesure où deux méthodes ayant le même nom et la même liste d'arguments ne peuvent différer par leur `return`.

Dans le code qui suit, la classe `FeuDeSignalisation` surcharge deux fois sa méthode `clignote()`, selon que l'on spécifie ou non dans les arguments les durées des phases allumées et éteintes.

```
class FeuDeSignalisation {
  void clignote() {
    System.out.println("premiere manière de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<3; j++)
         System.out.println("je suis eteint");
       for (int j=0; j<3; j++)
         System.out.println("je suis allume");
    }
  }
  void clignote(int a) {
    System.out.println("deuxieme maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
         System.out.println("je suis eteint");
```

```
        for (int j=0; j<a; j++)
          System.out.println("je suis allume");
      }
    }
    int clignote(int a, int b) {
      System.out.println("troisieme maniere de clignoter");
      for(int i=0; i<2; i++) {
        for (int j=0; j<a; j++)
            System.out.println("je suis eteint");
        for (int j=0; j<b; j++)
            System.out.println("je suis allume");
      }
      return b;
    }
    void clignote(int a, int b) {} /* Il est interdit de définir cette méthode, présentant
    ➥la même signature, mais un type de retour différent de la précédente */
}
```

La classe comme module fonctionnel

Différenciation des objets par la valeur des attributs

L'existence de la classe nous épargnera de préciser, pour chaque objet, le nombre et le type de ses attributs, ainsi que la signature et le corps des méthodes qui manipulent ces derniers, économie d'écriture non négligeable, et dont l'effet va croissant avec le nombre d'objets issus d'une même classe. En général, tout programme OO manipulera un grand nombre d'objets d'une même classe, comme des « voitures », des « feux » ou des « passants ». Ces objets seront stockés dans des ensembles informatiques particuliers, que l'on dénomme des *collections*. Il pourra s'agir d'ensembles extensibles, comme des listes, ou non, comme des tableaux. Les objets d'une même classe se différenciant entre eux uniquement par la valeur de leurs attributs, la seule information qu'il restera à préciser lors de la création de l'objet est, effectivement, la valeur initiale de ses attributs. C'est bien parce qu'il s'agit de l'unique information à compléter qu'une méthode particulière portant le même nom que la classe s'y emploiera. On appelle cette méthode singulière le *constructeur*.

Le constructeur

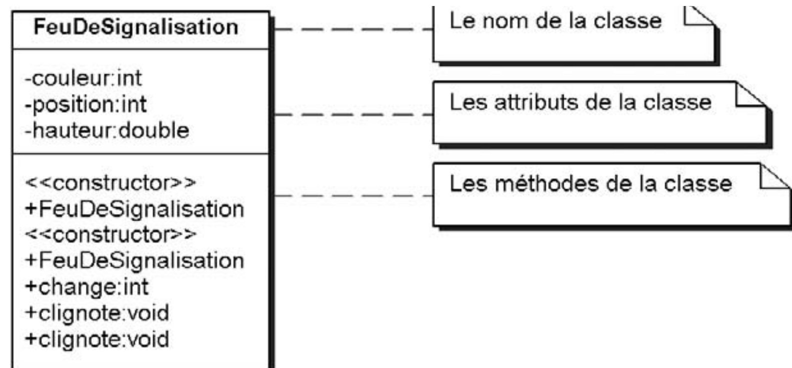


Figure 2-2

Addition de deux constructeurs surchargés dans la classe FeuDeSignalisation.

> **Le constructeur**
>
> Le constructeur est une méthode particulière, portant le même nom que la classe, et qui est définie sans aucun retour. Il a pour mission d'initialiser les attributs d'un objet dès sa création. À la différence des autres méthodes qui s'exécutent alors qu'un objet est déjà créé et sur celui-ci, il n'est appelé que lors de la construction de l'objet, et une version par défaut est toujours fournie par les langages de programmation. La recommandation, classique en programmation, est d'éviter de se reposer sur le « défaut », et, de là, toujours prévoir un constructeur pour chacune des classes créées, même s'il se limite à reproduire le comportement par défaut. Au moins, vous aurez « explicité » celui-ci. Le constructeur est souvent une des méthodes les plus surchargées, selon les valeurs d'attributs qui sont connues à la naissance de l'objet et qui sont passées comme autant d'arguments.

Ainsi le constructeur de la classe `FeuDeSignalisation` pourrait se définir comme suit :

```
FeuDeSignalisation(int positionInit, double hauteurInit) {
  /* pas de retour pour le constructeur */
  position = positionInit ;
  hauteur = hauteurInit ;
  couleur = 1 ;
}
```

Une surcharge de ce constructeur pourrait être imaginée (comme dans la figure 2-2), si seule la position était connue. Ce nouveau constructeur ne recevrait alors qu'un argument. Si aucun constructeur n'est spécifié dans la classe, un constructeur par défaut est fourni par les langages de programmation OO. Cependant, dès qu'un constructeur est défini dans la classe, et pour autant qu'il reçoive un ou plusieurs arguments, il ne sera plus possible de créer un objet en n'indiquant aucune valeur d'argument (sauf si le constructeur est explicitement surchargé par un autre qui ne reçoit aucun argument). Comme il est de bonne pratique en informatique de toujours avoir la maîtrise de l'initialisation des objets qu'on utilise, prenez l'habitude, pour éviter toute surprise, de toujours définir un constructeur. Muni de ce constructeur, l'instruction de toute création d'objet devrait maintenant vous paraître limpide :

```
FeuDeSignalisation unNouveauFeu = new FeuDeSignalisation(1, 3) ;
```

La dernière partie de l'instruction consiste en l'appel du constructeur. Notez que cette même instruction pourrait se décomposer en deux parties comme suit :

```
FeuDeSignalisation unNouveauFeu; // Création du seul référent initialisé à null
```

À la fin de cette première instruction, seul le référent est créé, créé et typé. Il n'est pas incorrect aux yeux du compilateur d'envoyer un message à même ce référent (comme `unNouveauFeu.change();`) bien que l'objet ne soit pas créé. Bien évidemment, cela plantera à l'exécution et produira une exception de type `NullPointer`, une des erreurs les plus fréquentes en Java et C# (et que le compilateur aussi attentif soit-il ne peut anticiper). Cela démontre que le compilateur ne s'intéresse jamais à la partie `new` des instructions de création d'objet et limite son attention à la déclaration statique. Le reste, la création à proprement parler, ne se déroule que pendant l'exécution.

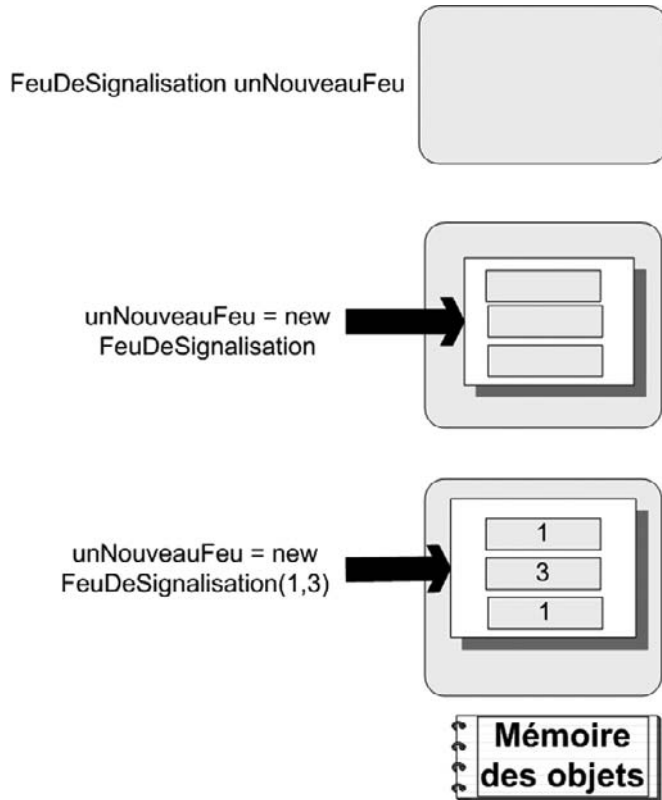
```
unNouveauFeu = new FeuDeSignalisation(1, 3); /* Création de l'objet et assignation de l'adresse de
l'objet comme valeur du référent */
```

La figure 2-3 illustre les trois étapes de la construction d'objet déclenchées par la seule instruction :

```
FeuDeSignalisation unNouveauFeu = new FeuDeSignalisation(1, 3) ;
```

Figure 2-3

Les trois étapes de la construction d'un objet par le truchement du « new ».



On peut légitimement se demander pourquoi est-il nécessaire d'indiquer deux fois le nom de la classe dans cette instruction, lors du typage de l'objet et lors de l'appel du constructeur. La raison en est très simple, mais il vous faudra attendre de comprendre le mécanisme d'héritage pour la découvrir. La classe renseignée à gauche pourrait être différente de la classe renseignée à droite. Plus précisément, le type de l'objet pourrait être une superclasse de la classe référée par le constructeur. Vous verrez dans quelques chapitres ce qu'est une superclasse et pourquoi peut différer le typage à gauche et à droite de l'instruction de création d'objet.

Mémoire dynamique, mémoire statique

À l'époque des tous premiers langages de programmation, toute réservation de l'espace mémoire nécessaire au stockage des données traitées par le programme, se faisait au départ du programme. À l'issue de la compilation, il y avait moyen de prévoir de quelle quantité de mémoire vive le programme aurait besoin pour son exécution. Lors de cette exécution, le programme se bornait à modifier les valeurs des variables stockées dans cette mémoire. Rien ne se créait et rien ne se perdait, du vrai « Lavoisier ». Ensuite, les langages ont autorisé l'allocation de mémoire au cours de l'exécution du code, mais toujours sous contrôle et dans un espace de

mémoire dédié et particulier dont la gestion s'effectue selon un principe dit de « mémoire pile » et toujours d'actualité dans la plupart des langages d'aujourd'hui. La gestion pile revient à empiler et dépiler les données à mémoriser en respectant un mécanisme de type *dernier rentrant premier sortant*, en fonction du début et de la fin des blocs d'instructions dans lesquels ces données opèrent. Ce mode de gestion mémoire est également décrit comme « statique ».

Le petit mot réservé, `new`, et bien connu des informaticiens, a chamboulé tout cela et est apparu le jour où ceux-ci ont accepté qu'un programme serait autorisé, au cours de son exécution, non seulement à allouer de l'espace mémoire pour y placer de nouvelles variables (ici, cela se réduit aux seuls objets). mais également de les disposer n'importe où dans cette mémoire et sans contraindre leur apparition et disparition de leur seule présence dans les blocs d'instructions. Ce mode alternatif de gestion mémoire est également taxé de « dynamique ».

Ainsi, lorsqu'en C++, digne héritier de cette tradition et langage capable des deux modes de gestion mémoire, la création d'un objet, instance de `FeuDeSignalisation`, se fait par la simple instruction suivante, en l'absence de `new` :

```
FeuDeSignalisation unNouveauFeu(1,3); // ici pas de classes à droite et à gauche !
```

Ce nouvel objet ne sera plus créé dans une zone mémoire, à découvrir pendant l'exécution (appelée *mémoire tas* et rediscutée plus en détail dans le chapitre 9), mais dans une zone mémoire identifiée pendant la compilation et gérée à la manière d'une *mémoire pile.* Dans de nombreux programmes, les objets apparaissent et disparaissent de manière non synchronisée à l'ouverture et la fermeture des blocs d'instructions, et établir l'espace mémoire au départ du programme à gérer de manière pile est, de fait, un peu trop contraignant. De plus, comme ces objets s'installent n'importe où dans la mémoire, il devient quasi impossible de les gérer à la manière d'une pile car, où se trouve le dessus de la pile, on vous le demande ? Mais, vous l'aurez compris, l'informatique est un sport extrême, et cette limitation fut levée par l'introduction du `new` et par l'existence des « référents ». Ces référents reçoivent comme valeur, dès la création de tout objet, l'adresse physique de ce dernier, quel que soit l'endroit où il se logera dans la mémoire. La disparition de cet objet est entièrement à repenser et le chapitre 9 s'y consacrera essentiellement.

La classe comme garante de son bon usage

Le fait que les objets ne puissent vivre sans leur classe, et que toute création et manipulation d'objet soient entièrement tributaires de ce qui est prévu dans sa classe, confère à la pratique de la programmation orientée objet une liste plutôt conséquente d'avantages. Tout d'abord, nous avons vu que la seule existence de la classe permet à tous les objets, sans que cela soit reprécisé pour chacun, de savoir automatiquement de quoi ils sont faits et ce qu'ils font. Ensuite, les informaticiens détestent les imprévus. Ils sont d'une susceptibilité telle qu'ils ne supportent pas, quand un programme s'exécute, que celui-ci ne se comporte pas comme prévu, ou, pire encore, comble de l'humiliation, que celui-ci « se plante ». Ils confient donc à un compilateur, à partir du logiciel qu'ils ont écrit dans un langage de programmation OO tel que C++, Smalltalk, Java ou C#, le soin de le traduire dans les instructions élémentaires du processeur (seul langage que le processeur comprend). Python et PHP 5 se singularisent ici (et ils se singulariseront encore souvent) car ce sont des langages dits de script, qui s'exécutent sans étape préalable de compilation afin de vérifier que le code est, dans son ensemble, correctement écrit. La traduction dans le langage du processeur se fait instruction par instruction et les instructions élémentaires qui sont produites sont exécutées au fur et à mesure.

En Java, C++ et C#, comme le compilateur a pour fonction critique de générer un code « exécutable », et que son utilisateur exige le moins inattendu possible, il prendra garde de vérifier que rien de ce qui est écrit par le programmeur ne puisse être source d’imprévu et d’erreur. Et c’est là que la classe joue de nouveau un rôle considérable, en permettant au compilateur de se transformer en un véritable cerbère, et de s’assurer que ce qui est demandé aux objets (essentiellement l’exécution de messages) est de l’ordre du possible. La classe est comme un texte contractuel. Elle disparaît lors de l’exécution pour donner place aux objets, tout en s’étant assurée par avance que tout de ce que feraient ses objets est conforme à ce qui est spécifié dans le contrat. Et ce contrat est passé avec le compilateur. On ne peut envoyer sur l’objet un message qui ne soit pas une des méthodes prévues par sa classe. On dit des langages qui permettent cette vérification, comme Java, C++ ou C#, qu’ils sont fortement typés. Le programmeur est à ce point contraint et tenu lors de l’écriture du logiciel (mais il faut croire qu’ils aiment ça) que, si ça passe à la compilation, il y a de fortes chances que cela passe aussi à l’exécution. Dans tous les cas, on n’est pas trop loin du but. De leur côté, Python et PHP 5, faisant l’impasse sur l’étape de compilation, laissent à l’exécution le soin de découvrir les instructions erronées, par le simple fait que celles-ci se planteront à l’exécution. Ne pas recourir à l’étape de compilation permet un gain indéniable en vitesse et en productivité, mais délègue à l’étape d’exécution (souvent critique) la responsabilité de repérer les dysfonctionnements. Malheureusement, à l’exécution, c’est parfois trop tard. Il n’y a plus grand-chose après ! C’est la différence entre s’informer au mieux sur la qualité d’un livre de programmation avant de l’acquérir et de le parcourir ou d’attendre d’ouvrir les premières pages pour se fixer les idées. Procéder sans vérification préalable permet d’aller plus vite mais… quelquefois au « casse-pipe ». Pour celui-ci, ça marche, mais c’est souvent risqué… reconnaissons-le.

Langage fortement typé

Un langage de programmation est dit fortement typé quand le compilateur vérifie que l’on ne fait avec les objets et les variables du programme que ce qui est autorisé par leur type. Cette vérification a pour effet d’accroître la fiabilité de l’exécution du programme. Java, C++ et C# sont fortement typés. L’étape de compilation y est essentielle. Ce n’est pas le cas, comme nous le verrons plus loin, de Python et PHP 5.

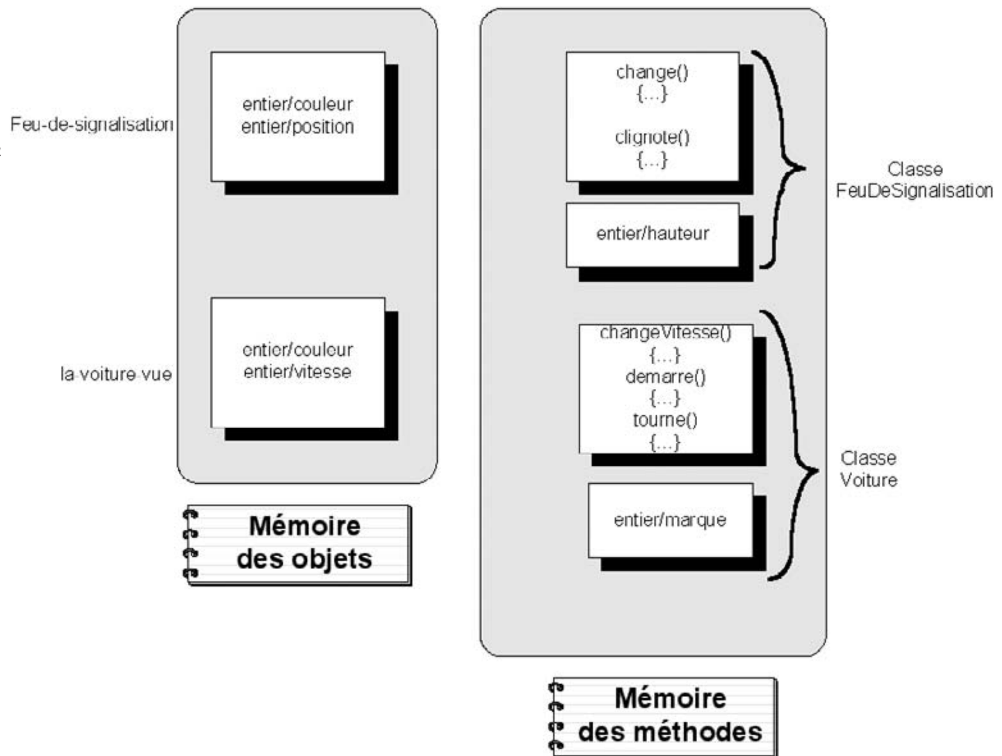
La classe comme module opérationnel

Mémoire de la classe et mémoire des objets

À propos des deux caractéristiques de la classe, on pourrait très synthétiquement dire de la première qu’elle est sa partie passive – représentée par les attributs qui sont associés aux objets (vu que chaque objet contiendra son propre ensemble d’attributs) –, et de la seconde qu’elle est sa partie active – représentée par les méthodes, en tant qu’associées à la classe, car les méthodes sont communes à tous les objets d’une même classe. Nous avons, dans le chapitre précédent, sciemment forcé cette séparation, en installant les attributs et les méthodes dans des espaces mémoires bien distincts.

Supposez maintenant que tous les feux de signalisation évoqués dans le chapitre précédent mesurent la même hauteur ou que, dans une application logicielle particulière, toutes les voitures soient de la même marque. Il n’est plus nécessaire d’installer ces deux attributs dans l’espace mémoire alloué à chaque objet, vu que leur valeur est commune à tous les objets. Il serait plus naturel, à l’instar des méthodes, de les installer dans les espaces mémoire dédiés aux classes. On qualifie ce type d’attribut particulier, dont les valeurs sont partagées par tous les objets et deviennent de ce fait plutôt attributs de classe que d’objet, d’*attribut statique.* Dans la figure 2-4, on retrouve ces attributs dans la zone mémoire adéquate.

Figure 2-4
Comment les attributs statiques « hauteur » de la classe FeuDeSignalisation et « marque » de la classe Voiture se retrouvent dans la mémoire associée aux classes et non plus aux objets.



Méthodes de la classe et des instances

Certaines méthodes peuvent également être déclarées statiques. Quel intérêt, alors, de forcer leur association à la classe plutôt qu'aux instances, constaterez-vous avec raison ? Les méthodes ne sont-elles pas toujours des méthodes de classe, ne le sont-elles par principe ? C'est très bien vu, mais vous vous serez rendu compte également que, bien qu'associée à la classe, toute méthode a, jusqu'à présent, toujours été exécutée sur un objet, ou simplement appelée à partir d'un objet, par une instruction semblable à `a.f(x)`, où `a` est le référent de l'objet, et `f(x)` la méthode. Une méthode statique aura la possibilité de pouvoir s'exécuter sans le moindre objet créé, uniquement à partir de sa classe, par une simple instruction comme `Class1.f(x)` (`Class1` étant le nom d'une classe) (nous verrons d'ailleurs qu'un langage comme C# n'accepte un appel de méthode statique qu'à partir de sa classe, ce qui est très logique). Une méthode statique pourra s'exécuter dès que la classe qui la contient est chargée en mémoire, y compris en l'absence de toute création d'objet. C'est par exemple le cas de toutes les méthodes mathématiques définies dans la classe Math en Java et C#, et qui s'appellent de la manière suivante : `Math.sin(45)` ou `Math.pow(2,1)`. On ne voit pas vraiment l'utilité de créer un objet de type `Math`.

Les praticiens de Java ou de C# connaissent tous, quelle que soit leur maîtrise de ces langages, une célèbre méthode statique, totalement inévitable, même par les plus novices d'entre eux: la méthode `main()`. La première méthode à s'exécuter lors du démarrage d'un programme se trouve définie, comme toute méthode (pas de traitement de faveur pour la « principale »), à l'intérieur d'une classe, mais une classe qui n'a pas forcément besoin de donner naissance à un objet. En C++, lourd tribut payé au C, et participant à rendre ce langage moins OO que les précédents, le « main » reste une procédure existant en dehors de toute classe. En Java et

C#, le « main » est une méthode statique, car il n'est, en effet, pas nécessaire de lancer cette méthode à partir d'un objet. Dans le cas contraire, il faudrait toujours s'assurer de la création d'un objet issu de la classe principale (qu'en général, rien ne force dans la conception de l'application) avant de déclencher le `main`. Mais qui pourrait s'en occuper sinon le `main` en effet ? Il s'en trouverait à se mordre la queue… Comme une méthode statique peut s'exécuter uniquement à partir de la classe, sans objet, les données que celle-ci manipulera se devront également de pouvoir exister sans objet. Toute méthode nécessite, lors de son exécution, l'adresse physique des données qu'elle manipule. Pour un objet, elle retrouve cette adresse à partir de son référent. Quand la méthode est statique, les données qu'elle utilise devront forcément se trouver dans l'espace mémoire réservé aux classes, et, par là même, se transformer en statique.

> **Statique**
>
> Les attributs d'une classe dont les valeurs sont communes à tous les objets, et qui deviennent ainsi directement associés à la classe, ainsi que les méthodes pouvant s'exécuter directement à partir de la classe, seront déclarés comme statiques. Ils pourront s'utiliser en l'absence de tout objet. Une méthode statique ne peut utiliser que des attributs statiques, et ne peut appeler en son sein que des méthodes également déclarées comme statiques.

Un premier petit programme complet dans les cinq langages

Ayant défini la manière de réaliser le `main`, nous avons tous les éléments en main (non, en main pas en `main`), pour réaliser un premier petit programme dans les cinq langages. Ce programme possédera une classe `FeuDeSignalisation`. Dans le `main`, il créera deux objets de cette classe, à l'aide de deux constructeurs surchargés. Il interrogera ensuite ces objets quant à la valeur de leur attribut statique `hauteur`, qu'il modifiera de plusieurs manières. Il finira par exécuter sur un de ces objets la méthode `clignote`, dans ses trois versions surchargées, passant comme argument les durées des phases éteintes et allumées. Pour l'instant, n'accordez aucune importance à la présence des mots-clés `public` et `private`, qu'il est nécessaire de spécifier, mais dont la signification sera longuement discutée dans la suite. Les cinq codes aboutissent au même résultat à l'écran (présenté sous le code Java).

En Java

Nous allons écrire le code Java dans un seul fichier, bien qu'il contienne deux classes et qu'une pratique bien meilleure consiste à séparer les classes par fichier. Nous le faisons ici pour des raisons de facilité et de simplicité au vu de la petitesse des codes présentés.

Le fichier Principale.java

```
/* Il est obligatoire en Java que la seule classe publique contenue dans le fichier porte le même
nom que celui-ci, ici Principale. C'est ce qui permet à Java de faire des liaisons dynamiques entre
les classes contenues dans des fichiers différents dès lors que chacun des fichiers ne contient
qu'une classe */

class FeuDeSignalisation {
  private int couleur;
  private int position ;
  private static double hauteur;  /* attribut statique */
```

```
  public FeuDeSignalisation(int couleurInit) { /* un premier constructeur */
    couleur = couleurInit;
    position = 0 ;
  }
  public FeuDeSignalisation(int couleurInit, double hauteurInit) { /* le constructeur est surchargé */
    couleur = couleurInit;
    hauteur = hauteurInit;
    position = 0 ;
  }
  public void setHauteur(double nouvelleHauteur) {
    hauteur = nouvelleHauteur;
  }
  public static void getHauteur() { /* méthode statique qui accède à l'attribut statique */
    System.out.println("la hauteur du feu est " + hauteur);
  }
  public void clignote() {
    System.out.println("premiere maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<2; j++)
           System.out.println("je suis eteint");
       for (int j=0; j<2; j++)
           System.out.println("je suis allume");
    }
  }
  public void clignote(int a) { // première surcharge de la méthode
    System.out.println("deuxieme maniere de clignoter");
    for(int i=0; i<2; i++) {
      for (int j=0; j<a; j++)
          System.out.println("je suis eteint");
      for (int j=0; j<a; j++)
          System.out.println("je suis allume");
    }
  }
  public int clignote(int a, int b) {
    System.out.println("troisieme maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
          System.out.println("je suis eteint");
       for (int j=0; j<b; j++)
         System.out.println("je suis allume");
    }
    return b;
  }
}
public class Principale {
  /* en Java, le fichier et la classe contenant le "main" doivent être appelés de la même façon */
  public static void main(String[] args) { /* c'est la manière d'écrire le main */
    FeuDeSignalisation unFeu = new FeuDeSignalisation(1,3.5); // création avec le deuxième
constructeur
    FeuDeSignalisation unAutreFeu = new FeuDeSignalisation(1); /* …avec le premier constructeur */
    unFeu.setHauteur(8.9) ;
```

```
    FeuDeSignalisation.getHauteur(); /* appel de la méthode statique à partir de la classe */
    unAutreFeu.setHauteur(10.6); /* tous les feux voient leur hauteur modifiée */
    unFeu.getHauteur(); /* appel de la méthode statique à partir de l'objet */
    System.out.println("**********  CLIGNOTEMENT **********");
    unFeu.clignote();
    unFeu.clignote(3);
    int b = unFeu.clignote(2,3);
  }
}
```

Résultats

```
la hauteur du feu est 8.9
la hauteur du feu est 10.6
**********  CLIGNOTEMENT ******
premiere maniere de clignoter
je suis eteint (écrit deux fois)
je suis allume (écrit deux fois)
je suis eteint (écrit deux fois)
je suis allume (écrit deux fois)
deuxieme maniere de clignoter
je suis eteint (écrit trois fois)
je suis allume (écrit trois fois)
je suis eteint (écrit trois fois)
je suis allume (écrit trois fois)
troisieme maniere de clignoter
je suis eteint (écrit deux fois)
je suis allume (écrit trois fois)
je suis eteint (écrit deux fois)
je suis allume (écrit trois fois)
```

EN C#

Le fichier Principal.cs

Le code C# est si proche du code Java que vous pourriez jouer au jeu des sept erreurs. D’ailleurs, il doit y en avoir moins. Parmi ces dernières: le `Main()` qui peut s’exécuter sans argument, les noms des méthodes débutant par une majuscule (dont le `Main`). C# choisit de débuter le nom des méthodes par une majuscule, contrairement à Java. Oui, on est d’accord, c’est un peu mesquin… Plus conséquent et plus logique (un bon point en faveur de C#), une méthode statique ne peut être appelée qu’à partir de sa classe et non plus à partir de ses instances (Java et C++ offrent les deux possibilités). Enfin, si vous ajoutez par mégarde un *return* devant le constructeur, tout comme en C++, cela provoquera une erreur lors de la compilation. En Java, vous aurez juste déclaré une nouvelle méthode, qui joue un rôle autre que celui de constructeur.

```
/* Bien que cela soit une excellente habitude surtout dans le cas recommandé où vous n'installez
qu'une classe par fichier, C# n'oblige pas, comme Java, à donner au fichier le même nom que la
classe qu'il contient */

using System; /* comme nous utiliserons dans le code l'instruction « Console.WriteLine », celle-ci
➥se trouve dans l'assemblage « System » qu'il est nécessaire de spécifier */
```

```
class FeuDeSignalisation {
  private int couleur;
  private int position ;
  private static double hauteur;

  public FeuDeSignalisation(int couleurInit) {
    couleur = couleurInit;
      position = 0 ;
  }
  public FeuDeSignalisation(int couleurInit, double hauteurInit) {
    couleur = couleurInit;
    hauteur = hauteurInit;
    position = 0 ;
  }
  public void setHauteur(double nouvelleHauteur) {
    hauteur = nouvelleHauteur;
  }
  public static void getHauteur()
  {
    Console.WriteLine("la hauteur du feu est " + hauteur); /* la manière d'écrire sur l'écran en C# */
  }
  public void clignote() {
    Console.WriteLine("premiere maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<2; j++)
          Console.WriteLine("je suis eteint");
       for (int j=0; j<2; j++)
          Console.WriteLine("je suis allume");
    }
  }
  public void clignote(int a) {
    Console.WriteLine("deuxieme maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
          Console.WriteLine("je suis eteint");
       for (int j=0; j<a; j++)
          Console.WriteLine("je suis allume");
    }
  }
  public int clignote(int a, int b) {
    Console.WriteLine("troisieme maniere de clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
           Console.WriteLine("je suis eteint");
       for (int j=0; j<b; j++)
           Console.WriteLine("je suis allume");
    }
    return b;
  }
}
```

```
public class Principale {
  public static void Main() { /* voici le Main en C# */
    FeuDeSignalisation unFeu = new FeuDeSignalisation(1,3.5);
    FeuDeSignalisation unAutreFeu = new FeuDeSignalisation(1);
    unFeu.setHauteur(8.9) ;
    FeuDeSignalisation.getHauteur();
    unAutreFeu.setHauteur(10.6);
    /* unFeu.getHauteur(); impossible en C# */
    Console.WriteLine("**********  CLIGNOTEMENT **********");
    unFeu.clignote();
    unFeu.clignote(3);
    int b = unFeu.clignote(2,3);
  }
}
```

En C++

Le fichier Principal.cpp

En C++, de très nombreuses différences apparaissent. Dans la suite, nous aurons l'occasion de revenir sur nombre d'entre elles. Parmi les plus notables, `main` est une procédure ou une fonction, mais plus une méthode. Elle peut ou non retourner quelque chose. Dans le code, nous supposons qu'elle peut retourner, comme il est classique en C++, un code d'erreur si quelque chose se passe mal lors de l'exécution du programme.

```
#include "stdafx.h"
#include "iostream.h"  /* afin de pouvoir utiliser le cout */
class FeuDeSignalisation {
private: /* le public et le private sont mis en évidence */
  int couleur;
  int position;
  static double hauteur;
public:
FeuDeSignalisation (int couleurInit) {
    couleur = couleurInit;
     position = 0 ;
}
FeuDeSignalisation (int couleurInit, int positionInit):couleur(couleurInit),position(positionInit) {
 /* le constructeur peut initialiser les attributs directement à partir de
  * la déclaration de sa signature */
}
void setHauteur(double nouvelleHauteur) {
    hauteur = nouvelleHauteur;
}
void static getHauteur() {
    cout << "la hauteur du feu est " << hauteur << endl; /* la manière d'écrire sur l'écran en C++ */
}
void clignote() {
    cout <<"premiere maniere de clignoter"<< endl;
    for(int i=0; i<2; i++) {
       for (int j=0; j<2; j++)
           cout << "je suis eteint" << endl;
```

```
        for (int k=0; k<2; k++)
            cout <<"je suis allume" << endl;
    }
}
void clignote(int a) {
    cout << "deuxieme maniere de clignoter" << endl;
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
            cout <<"je suis eteint" << endl;
       for (int k=0; k<a; k++)
            cout <<"je suis allume" << endl;
    }
}
int clignote(int a, int b) {
    cout << "troisieme maniere de clignoter" << endl;
    for(int i=0; i<2; i++) {
       for (int j=0; j<a; j++)
            cout << "je suis eteint" << endl;
       for (int k=0; k<b; k++)
            cout <<"je suis allume" << endl;
    }
    return b;
  }
};

double  FeuDeSignalisation::hauteur = 3.5;  /* C'est l'unique manière d'initialiser la valeur de
➥l'attribut statique*/

int main(int argc, char* argv[]) {
/* On crée un objet sur la mémoire pile */
  FeuDeSignalisation unFeu (1, 3);
/* On crée un objet avec " new " sur la mémoire tas */
  FeuDeSignalisation *unAutreFeu = new FeuDeSignalisation(1);
  unFeu.setHauteur(8.9);
/* L'unique manière d'évoquer la méthode statique, quand on le fait à partir de la classe */
  FeuDeSignalisation::getHauteur();
  unAutreFeu->setHauteur(10.6); /* le point se transforme en flèche pour des objets sur le tas */
  unAutreFeu->getHauteur();
  cout <<  "**********  CLIGNOTEMENT **********" << endl;
  unFeu.clignote();
  unFeu.clignote(3);
  int b = unFeu.clignote(2,3);
  return 0;
}
```

C++ autorise l'hybridation des deux modes de programmation : procédural et objet, et, pour le `main`, on n'a pas vraiment le choix. La référence à une classe se fait toujours par *`Classe::méthode`* comme, dans le code, pour l'appel de la méthode statique, quand cet appel se fait à partir de la classe. Les objets peuvent être créés sur la pile, sans le `new`, ou dans le tas, avec le `new`. Lorsqu'ils sont créés dans le tas, les objets sont alors adressés

par une variable de type pointeur, faisant ici office de référent (nous reviendrons largement sur la gestion mémoire dans le chapitre 9).

Pointeur

Un pointeur est une variable dont la valeur, comme le référent, est l'adresse d'une autre variable. Il fonctionne par adressage indirect. En C++, les pointeurs ne sont pas typés comme les référents. On peut par exemple les traiter comme des entiers, les incrémentant ou les décrémentant. Si peu contrainte, leur utilisation comporte de nombreux risques, car on voyage dans la mémoire sans le filet de sécurité assuré par les langages plus typés. Le typage plus strict des référents, en Java et en C#, les force à ne pointer, toujours, que sur des objets existants, d'une classe donnée.

L'évocation des méthodes sur le pointeur se fait en remplaçant le point par la flèche. Dans la procédure `main`, les deux objets, l'un dans la mémoire pile (qu'on associe aux méthodes avec le « . ») et l'autre dans la mémoire tas (qu'on associe aux méthodes avec le « -> »), sont utilisés, par la suite, de manière indifférenciée.

En Python

Le fichier Principal.py

```
class FeuDeSignalisation:
        __couleur=0  #il s'agit d'office d'un attribut de la classe
        __position=0 #donc statique
        __hauteur=0  #ici aussi statique

        def __init__(self, couleurInit):
                self.__couleur=couleurInit

        #L'utilisation de self indique que l'on peut également
        #référencer un attribut à partir d'un objet.
        #C'est en utilisant self qu'un attribut de classe
        #deviendra ici un attribut d'objet.

        #Une première manière très simple de réaliser un constructeur capable d'une certaine forme
        #de surcharge est :

          def __init__(self, couleurInit=None,hauteurInit=None):
                  self.__couleur=couleurInit
                  self.__hauteur=hauteurInit

         #Une autre manière détournée mais plus sophistiquée d'opérer une surcharge.
         #On redéfinit une méthode statique pouvant faire office de constructeur.

        def other__init(couleurInit, hauteurInit):
                result=FeuDeSignalisation(couleurInit)
                result.__couleur=couleurInit
                result.__hauteur=hauteurInit
                return result

        #On en fait une méthode statique car elle ne peut s'appeler
        #qu'à partir de la classe.

        other__init=staticmethod(other__init)
```

```
        def setHauteur(self,nouvelleHauteur):
                FeuDeSignalisation.__hauteur=nouvelleHauteur
        def getHauteur():
                print "la hauteur du feu est %s" % (FeuDeSignalisation.__hauteur)

   #Ici également on transforme cette méthode en statique.

        getHauteur = staticmethod(getHauteur)

        #Pas de surcharge, ici aussi on s'arrange comme on peut.

        def clignote(self,a='omitted',b='omitted'):
                if a == 'omitted' and b == 'omitted':
                        print ("premiere maniere de clignoter")
                        i=0
                        while i<2:
                                j=0
                                while j<2:
                                        print "je suis eteint"
                                        j+=1
                                j=0
                                while j<2:
                                        print "je suis allume"
                                        j+=1
                                i+=1
                elif a!='omitted' and b=='omitted':
                        print ("deuxieme maniere de clignoter")
                        i=0
                        while i<2:
                                j=0
                                while j<a:
                                        print "je suis eteint"
                                        j+=1
                                j=0
                                while j<a:
                                        print "je suis allume"
                                        j+=1
                                i+=1
                else:
                        print ("troisieme maniere de clignoter")
                        i=0
                        while i<2:
                                j=0
                                while j<a:
                                        print "je suis eteint"
                                        j+=1
                                j=0
                                while j<b:
                                        print "je suis allume"
                                        j+=1
                                i+=1
                        return b

#Ce n'est pas vraiment l'appel du constructeur
#bien qu'il s'agisse effectivement d'une initialisation
#de l'objet.
```

```
unFeu=FeuDeSignalisation.other__init(1,3.5)

#Par le premier type de surcharge du constructeur, on aurait également pu directement et plus
#simplement faire appel au constructeur officiel
#unFeu=FeuDeSignalisation(1,3.5)


#Ici, c'est bien l'appel de ce constructeur
#à proprement parler.

unAutreFeu=FeuDeSignalisation(1)

unFeu.setHauteur(8.9)
FeuDeSignalisation.getHauteur()
unAutreFeu.setHauteur(10.6)
unFeu.getHauteur()
print "****** CLIGNOTEMENT ******"
unFeu.clignote()
unFeu.clignote(3)
b=unFeu.clignote(2,3)
```

Python est un langage ayant recherché dès son origine une grande simplicité d'écriture, tout en conservant tous les mécanismes de programmation OO de haut niveau. Il cherche à soulager au maximum le programmeur de problèmes syntaxiques non essentiels aux fonctionnalités clés du programme. Les informaticiens parlent souvent à son compte d'un excellent langage de prototypage qu'il faut remplacer par un langage plus « solide » tel Java, les langages .Net ou C++, lorsqu'on arrive aux termes de l'application. Sa syntaxe de base, par les raccourcis qu'elle autorise, est donc assez différente de celle des trois autres langages. Ainsi, par rapport aux langages précédents, une surprise de taille nous attend, surtout pour ceux qui ont vu passer des kyrielles de langages de programmation : les accolades et les points-virgules ont disparu. Python détecte les limites des blocs d'instructions grâce à l'indentation des lignes, indentation qui devient dès lors capitale. Cette nouvelle règle syntaxique a fait d'une pratique souvent recommandée une obligation. Python est un langage OO, mais tout comme C++, il n'oblige pas à la pratique OO. En témoigne ici l'absence d'une classe principale et même de la méthode main, car il suffit d'écrire le programme appelant au même niveau que la définition des classes. Reconnaissez que, de la sorte, un programme est bien plus simple à démarrer que par le très laborieux `public void static main (String[] args)` de Java. Le sempiternel `hello world` se fait simplement par `print "hello world"`. Tentez de faire plus simple et vous conviendrez aisément des ambitions pédagogiques de Python.

Une autre différence essentielle, visible dans la déclaration des attributs, est que Python n'est pas un langage typé, du moins en ce qui concerne les variables et les attributs. Il est dit « typé dynamiquement » en ce sens que le type de la variable est alloué en fonction de ce qu'elle contient et peut ainsi changer au fil des affectations. C'est une pratique très discutable qui permet une économie d'écriture, mais peut occasionner quelques comportements indésirables lors de l'exécution. Comme nous avons déjà eu l'occasion de le dire, le rôle du typage est inséparable de celui du compilateur et permet à ce dernier de détecter des erreurs de substitution entre variables. Python ne compilant pas, ce typage explicite ne s'avère plus autant nécessaire, au risque que certains problèmes ne se produisent qu'à l'exécution. Pour déclarer un attribut privé, il suffit de faire précéder son nom de deux underscores. Une faiblesse additionnelle est que Python ne supporte pas la surcharge de méthodes. C'est assez compréhensible vu l'absence de typage explicite ; difficile de distinguer deux signatures de méthode par le type de leurs arguments. Dans le code ci-dessus, certaines astuces sont adoptées pour contourner cette limitation. Le constructeur doit avoir le nom de `__init__`. Nous discuterons du `self`, indispensable dans la définition des méthodes s'exécutant à partir des objets (c'est-à-dire non statiques), par la suite.

En PHP 5

Le fichier Principal.php

```
<html>
<head>
<title> Classe Feu de Signalisation </title>
</head>
<body>
<h1> Classe feu de signalisation </h1>
<br>

<?php
   class FeuDeSignalisation {
          private $couleur; // toute variable en PHP débute par $
          private $position;
          private static $hauteur;


           public function __construct() { /* définition d'un constructeur surchargeable assez
           proche de Python*/
                      $num_args=func_num_args();
                      switch ($num_args)
             {

                      case '0':
                           $this->couleur = $this->position = 0; /* $this est indispensable
                           pour les attributs d'objet*/
                           self::$hauteur = 0; /* self l'est pour les attributs statiques */
                           break;
                           case '1':
                                     $this->couleur = func_get_arg(0);
                                     $this->position = 0;
                                     self::$hauteur = 0;
                                     break;
                           case '2':
                                     $this->couleur =func_get_arg(0);
                                     $this->position = 0;
                                     self::$hauteur=func_get_arg(1);
                                     break;
                           }
              }

              public function setHauteur($nouvelleHauteur) {
                         self::$hauteur = $nouvelleHauteur;
              }

              static public function getHauteur() {
                            print ("la hauteur du feu est ". self::$hauteur . "<br>\n");
              }
```

```
              public function clignote() {  // définition d'une méthode clignote surchargeable
                            $num_args=func_num_args();
                            $b=0;
                            switch ($num_args)
                            {

                            case '0':
                                      print("premiere maniere de clignoter <br>\n");
                                      for ($i=0;$i<2;$i++){
                                                for ($j=0;$j<2;$j++)
                                                          print("je suis eteint <br>\n");
                                                for ($j=0;$j<2;$j++)
                                                          print("je suis allume <br>\n");
                                       }
                                       break;
                            case '1':
                                       print("deuxieme maniere de clignoter <br>\n");
                                       $a=func_get_arg(0);
                                       for ($i=0;$i<2;$i++){
                                                for ($j=0;$j<$a;$j++)
                                                          print("je suis eteint <br>\n");
                                                for ($j=0;$j<$a;$j++)
                                                          print("je suis allume <br>\n");
                                        }
                                        break;
                            case '2':
                                       print("troisieme maniere de clignoter <br>\n");
                                       $a=func_get_arg(0);
                                       $b=func_get_arg(1);
                                       for ($i=0;$i<2;$i++){
                                                for ($j=0;$j<$a;$j++)
                                                          print("je suis eteint <br>\n");
                                                for ($j=0;$j<$b;$j++)
                                                          print("je suis allume <br>\n");
                                        }
                                        return $b;
                            }
              }
}

$unFeu = new FeuDeSignalisation(1,3.5);
$unAutreFeu = new FeuDeSignalisation(1);
$unFeu->setHauteur(8.9);
FeuDeSignalisation::getHauteur();
$unAutreFeu->setHauteur(10.6);
$unFeu->getHauteur();
print("********** CLIGNOTEMENT ******** <br>\n");
$unFeu->clignote();
$unFeu->clignote(3);
$b=$unFeu->clignote(2,3);
?>

</body>
</html>
```

PHP 5 est devenu le langage de prédilection pour nombre de maîtres toileurs (webmestres) qui lui trouvent de nombreux avantages pour la conception de sites web dynamiques. PHP est un langage d’écriture de script qui s’exécute sur un serveur web et permet de mêler assez simplement les informations de structuration d’un site web (exprimé dans le langage HTML) et les instructions de programmation permettant de rendre ce même site dynamique et interactif. Créé en 1995, il est devenu, dans sa cinquième version (début des années 2000), pleinement orienté objet, ce qui a contribué davantage encore à son succès et l’a rendu responsable du bon fonctionnement et de l’attrait de plusieurs millions de sites web. Ceux-ci également, comme tout type de logiciel aujourd’hui, tentent à s’enrichir de plus en plus de nombreuses fonctionnalités : complexification croissante, dont la programmation et la modularisation OO contribuent à adoucir les effets dévastateurs sur la santé mentale des programmeurs. Ce livre n’a en aucun cas l’ambition d’aborder, même un tant soi peu, la conception de sites web. Le sujet s’avère suffisamment riche pour dédier un livre sinon une librairie entière à son seul traitement. Ne nous intéressera donc dans PHP que le côté « langage de programmation OO », et nullement la manière dont il s’harmonise avec les informations de structuration et de contenu propres à tout site web. On considérera aussi comme résolus, tous les problèmes d’installation de serveur web indispensables à l’utilisation et au bon fonctionnement du PHP. Néanmoins, le cadre d’exécution des scripts PHP étant un navigateur Internet, le code PHP 5 se trouve forcément, comme dans l’exemple précédent, imbriqué dans un environnement HTML. À la suite de quelques instructions HTML, le code débute par la balise `<?php` et se termine par la balise `?>`. Le reste devrait vous paraître assez familier. Étant conçu comme un langage de script, PHP se passe, tout comme Python (c’est une espèce d’hybride entre la famille C++/Java et les langages de script tels Perl ou Python), de compilateur et de typage explicite (avec les mêmes avantages et inconvénients épinglés pour Python). Si vous avez « encaissé » les codes Java et Python, celui du PHP 5 ne devrait pas vous poser de problème particulier, sinon que vous dire…. Retour à la case Java !

La classe et la logistique de développement

Classes et développement de sous-ensembles logiciels

Nous aurons souvent l’occasion de revenir sur l’avantage suivant: la classe permet un découpage logiciel des plus naturels. Un programme, quel qu’il soit, se compose toujours d’une structure de données et d’un ensemble d’opérations portant sur ces données. Or, un programme, cela peut devenir bien vite très gros, des millions de lignes de code dit-on pour Windows (bien que nous reconnaissions ne pas les avoir comptées). La préoccupation pour le programmeur ou, plus souvent, l’équipe de programmeurs devient de trouver un moyen simple et naturel de découper le programme en un ensemble de modules gérables et suffisamment indépendants entre eux. Vous nous voyez venir avec nos gros sabots. Mais oui, bien sûr, pourquoi ne pas découper tout le logiciel en ses classes, puisque chacune d’entre elles, tout comme un petit programme à part entière, se retrouve avec sa structure de données et ses opérations? Pour l’informaticien quelles équations de rêve que les suivantes : une classe = un type = un module = un fichier, un programme = un ensemble de classes en interaction = un ensemble de fichiers automatiquement liés. C’est donc autour de la classe que l’informaticien, idéalement, tracera les traits pointillés qui lui permettront de découper son code en fichiers. C’est, parmi tous les langages que nous découvrons dans ce livre, Java qui a poussé cette logique à son paroxysme, en insistant pour placer une classe par fichier (si vous ne le faites pas, il le fait pour vous à la compilation) et en donnant au fichier le même nom que celui de sa classe.

Classes, fichiers et répertoires

De même que vous organisez l'emplacement et la gestion des fichiers à l'aide de répertoires imbriqués selon les thèmes repris par ces fichiers, les classes pourront également être organisées en assemblage, selon, là encore, de simples critères sémantiques. Les classes portant sur un domaine semblable se regrouperont dans un même assemblage. L'organisation des classes en assemblage sera transposée de manière isomorphe dans une organisation des fichiers qui contiennent ces classes en répertoire. Les assemblages s'organiseront entre eux, tous comme les répertoires, de manière hiérarchique. Toute dépendance entre classes par l'envoi de message débouchera sur une liaison des plus simples à mettre en œuvre entre les fichiers qui contiennent ces classes. Aucune liaison dynamique, autre que celle directement prévue par les déclarations des classes, n'apparaîtra comme nécessaire. Idéalement, si un objet de la classe `A` nécessite de connaître la classe `B` pour s'exécuter, cela sera inscrit noir sur blanc dans le code et n'appellera aucune instruction additionnelle, au niveau du système d'exploitation, pour relier les deux fichiers.

Liaison naturelle et dynamique des classes

La classe, par le fait qu'elle s'assimile à un petit programme à part entière, constitue un module idéal pour le découpage du logiciel en ses différents fichiers. La liaison sémantique entre les classes, rendue possible si la première intègre en son code un appel à la seconde, devrait suffire à relier de façon dynamique, pendant la compilation et l'exécution du code, les fichiers dans lesquels ces classes sont écrites. C'est principalement dans Java que cette logique de découpe et d'organisation sémantique du code en ses classes isomorphes à la découpe et l'organisation physique en fichiers sont le plus scrupuleusement forcées par la syntaxe. C'est un très bon point en faveur de Java.

Ces différents rôles, endossés par les classes, ont été disséqués en profondeur par Bertrand Meyer dans son remarquable ouvrage *Conception et programmation orientées objet*. Citons-le : « Dans les approches non OO, les concepts de module et de type restent distincts. La propriété la plus remarquable de la notion de classe est qu'elle généralise ces deux concepts, les fusionnant en une seule construction logique. Une classe est un module, ou une unité de décomposition logicielle ; mais c'est aussi un type… ».

Bertrand Meyer

Bertrand Meyer est un personnage incontournable du monde de l'OO, une de ses plus grosses pointures. Formé en France, il se partage ces dernières années entre la Suisse (il est professeur à l'ETH de Zurich), les États-Unis et l'Australie. Il est toujours extrêmement actif, et, plus récemment, s'est beaucoup investi dans la plate-forme .Net de Microsoft. Pour nous, ici, il est surtout le père du langage de programmation Eiffel (il a fondé la compagnie Eiffel Software à Santa Barbara en Californie), un langage OO clef qui, même s'il ne rivalisera sans doute jamais, en termes de popularité, avec Java, C++ et C#, est une espèce d'idéal à atteindre par tous les langages OO. On le retrouve d'ailleurs intégré dans .Net. On trouve dans Eiffel toutes les bonnes choses de l'OO, le tout objet, l'encapsulation, le polymorphisme, l'héritage simple et multiple, la généricité, le ramasse-miettes…, et plus encore. Ce langage est décrit en profondeur dans une deuxième réalisation remarquable de Meyer, l'ouvrage *Conception et programmation orientées objet* (Eyrolles), qui en est à sa deuxième édition. C'est une référence indispensable pour qui cherche à s'aventurer au plus profond dans les questions et les méandres de la pratique OO. Les réponses que vous ne trouverez pas ici, vous devriez, au risque d'une lecture un peu plus corsée (on n'a rien sans mal), les trouver dans l'ouvrage de Meyer. Ce livre n'est pas toujours d'un abord facile, mais l'effort déployé à comprendre ce qui y est dit est souvent très gratifiant.

Depuis plusieurs années, Meyer essaie d'imposer une vision à la fois très personnelle et très formelle de la programmation OO, qui rajoute à toutes les bonnes choses déjà connues et reprises dans son ouvrage, ainsi que dans le nôtre, la mise en pratique de la « conception par contrat ». Très schématiquement, si les classes sont à même de fournir des prestations pour des clients, elles devraient le faire sous une forme de contrat, clairement établi et explicité.

> Pour autant que soit garanti un ensemble de pré-conditions nécessaires à la bonne exécution du contrat, dans ce dernier la classe prestataire du service s'engage à fournir un ensemble de post-conditions remplissant les attentes du client. De plus, chaque classe a la responsabilité personnelle de préserver son intégrité, en respectant, quoi qu'elle fasse, un ensemble d'invariants. Ce qu'il faut comprendre ici, c'est que, même si cette pratique peut être, à coup de triturations d'écriture, implémentée dans tous les langages, Meyer estime que ces invariants devraient plus intimement et plus naturellement être intégrés dans la syntaxe de ces langages, ce qu'Eiffel accomplit (et le « tour » est joué). La mise en pratique de ces différentes additions (pré et post conditions ainsi que les invariants) devrait assurer le développement de logiciels plus fiables et plus faciles à maintenir et à faire évoluer.
>
> En décembre 2005, Bertrand Meyer fut victime d'une plaisanterie assez macabre qui pourrait provenir de l'un de ses étudiants. L'encyclopédie online Wikipédia annonça son décès (le lendemain de la publication des résultats d'un de ses examens au polytechnique de Zurich), et il fallut quelques jours pour corriger cette fausse mauvaise nouvelle. Cela permit à certains de dénoncer le fonctionnement et l'existence même de Wikipédia qui avait pourtant trouvé en Bertrand Meyer l'un de ses défenseurs les plus ardents. Version web de l'arroseur arrosé.

Exercices

Exercice 2.1

Réalisez la classe `voiture` avec un changement de vitesse, en y installant, tout d'abord, deux méthodes ne retournant rien, mais permettant, l'une d'incrémenter la vitesse, et l'autre de décrémenter la vitesse. Surchargez ensuite la méthode d'incrémentation de vitesse, en lui passant, en argument, le nombre de vitesses à incrémenter.

Exercice 2.2

Soit la déclaration de la méthode suivante :

```
public void test(int a) {}
```

Quelles sont les surcharges admises entre ces différentes possibilités ?

```
public void test() {}
public void test(double a) {}
public void test(int a, int b) {}
public int test(int a) {}
```

Exercice 2.3

Les fichiers Java, `A.java`, et C#, `A.cs`, suivants ne compileront pas, et ce, malgré leur grande ressemblance, pour des raisons différentes. Expliquez pourquoi.

A.java

```
public class A {
  void A(int i) {
    System.out.println("Hello");
  }
  public static void main(String[] args) {
    A unA = new A(5);
  }
}
```

A.cs

```
using System;
public class A {
  void A(int i) {
    Console.WriteLine("Hello");
  }
  public static void Main() {
    A unA = new A(5);
  }
}
```

Exercice 2.4

Réalisez le constructeur de la classe `voiture`, initialisant la vitesse à 0. Surchargez ce constructeur si l'on connaît la vitesse initiale.

Exercice 2.5

Parmi les attributs suivants de la classe `Renault Kangoo`, la version avec toutes les options possibles, séparez ceux que vous déclareriez comme statiques des autres : vitesse, nombre de passagers, vitesse maximale, nombre de vitesses, capacité du réservoir, âge, puissance, prix, couleur, nombre de portières.

Exercice 2.6

Les trois codes suivants ne trouveront pas grâce aux yeux du constructeur. Une seule erreur s'est glissée dans chacun d'eux. Corrigez-les.

Code 1 : Fichier Java : PrincipalTest.java

```
class Test {
  int a;
  Test (int b) {
    a = b;
  }
}

public class PrincipalTest {
  public static void main(String[] args) {
    Test unTest = new Test();
  }
}
```

Code 2 : Fichier C# : PrincipalTest.cs

```
class Test {
  private int a;
  private int c;
  public Test (int b) {
    a = b;
  }
  public Test (int e, int f) {
      a = e;
      c = f;
  }
}

public class PrincipalTest {
  public static void Main(){
    Test unTest = new Test(5);
    Test unAutreTest = new Test(5, 6.5);
  }
}
```

Code 3 : Fichier C++ : PrincipalTest.cpp

```
#include "stdafx.h"
class Test {
private:
  int a, b;

public:
  Test (int c, int d) {
    a = c;
    b = d;
  }
  Test (int c):a(c) {}
};

int main(int argc, char* argv[]) {
  Test unTest(5);
  Test *unAutreTest = new Test(6,10);
```

```
  Test unTroisiemeTest;
  return 0;
}
```

Exercice 2.7

Les trois codes suivants ne trouveront pas grâce aux yeux du compilateur. Une seule erreur s’est glissée dans chacun d’eux. Corrigez-les.

Code Java : fichier PrincipalTest.java

```
class Test {
  int a;
  int c;

  Test (int b) {
    a = b;
  }
  static int donneC() {
    return c;
  }
}

public class PrincipalTest {
  public static void main(String[] args) {
    Test unTest = new Test(5);
  }
}
```

Code C# : fichier PrincipalTest.cs

```
class Test {
  private int a;
  static private int c;

  public Test (int b) {
    a = b;
  }
  public Test (int e, int f) {
     a = e;
     c = f;
  }
  public static int donneC() {
    return c;
  }
}
public class PrincipalTest {
  public static void Main() {
    Test unTest = new Test(5);
    unTest.donneC();
  }
}
```

Code C++ : PrincipalTest.cpp

```
#include "stdafx.h"
class Test {
private:
  int a, b;
  static int c;
public:
  Test (int e, int f) {
     a = e;
     c = f;
  }
  Test (int e):a(e) {}

  static int donneC() {
    return c;
  }
};
int main(int argc, char* argv[]) {
  Test unTest(5);
```

```
    Test *unAutreTest = new Test(6,10);
    unAutreTest->donneC();
    return 0;
}
```

Exercice 2.8

Réalisez en Java et en C#, un programme contenant une classe `Point`, avec ses trois coordonnées dans l’espace x,y,z, et que l’on peut initialiser de trois manières différentes (selon les valeurs initiales connues des trois coordonnées, on connaît soit x, soit x et y , soit x et y et z). Ensuite, intégrez dans la classe une méthode `translate()` qui est surchargée trois fois, dépendant également desquelles des trois valeurs des translations sont connues.

Exercice 2.9

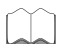
Créez deux objets de la classe `Point` à peine réalisée, et testez le bon fonctionnement du programme quand vous translatez ces points.

3

Du faire savoir au savoir-faire… du procédural à l'OO

Ce chapitre distingue l'approche dite procédurale, axée sur les grandes activités de l'application, de l'approche objet, axée sur les acteurs de la simulation et la manière dont ils interagissent. Nous illustrons cette distinction à l'aide de la simulation d'un petit écosystème.

Sommaire : Procédural *versus* OO — Activité *versus* acteurs — Dépendance fonctionnelle mais indépendance dans le développement — Introduction à la relation interclasses ou client-fournisseur — Acteurs collaborant



Candidus – Bon ! Maintenant qu'on a donné des jouets à bébé, il faut lui expliquer comment tout cela fonctionne. J'aurais envie de mettre tout à sa portée, bien rangé sur la table, mais ne va-t-il pas tout mélanger ?

Doctus – Mieux vaut lui présenter chacun des petits puzzles l'un après l'autre.

Cand. – Si je comprends bien, tu veux lui faire construire un gros truc sans qu'il s'en rende compte. Pourtant je rêve d'un ordinateur qui me comprendrait à demi-mot !

Doc. – Je propose de diviser une structure complexe en sous-ensembles simples. Par exemple, selon la méthode classique, nos voitures se présentaient sous forme de pièces détachées devant être assemblées et contrôlées et c'était à toi d'en vérifier l'intégrité avant chaque usage. Avec l'OO, ta voiture est toujours prête, tu montes dedans et c'est parti !

Cand. — Normalement, c'est le programmeur qui se charge de réaliser ce qui a été envisagé lors de la phase de conception…

Doc. — Oui, mais la programmation objet a un rôle à jouer autant à la phase d'analyse, qu'à la conception et même à l'exécution. Les pièces de notre puzzle ne sont plus de simples morceaux de carton, elles participent activement à notre jeu ! Chaque pièce sera indissociable de son mode d'emploi !

Cand. — Tu veux dire que les données et les fonctions seront scotchées les unes aux autres ? Mais où vais-je donc pouvoir mettre mes *goto* alors ? Et que deviennent les procédures de notre programme ?

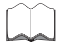
Doc. – Il s'agit pour schématiser de les remplacer par un jeu de transactions entre les différents acteurs.

Cand. — Ça c'est fort ! Les jouets vont jouer les uns avec les autres ! Mais comment faire, lorsqu'il y a plusieurs fabrications de jouets, pour créer des interfaces ? C'est sûr que les cotes et les formes des pièces du puzzle devront être bien définies pour que bébé puisse jouer sans s'énerver.

Doc. – Chaque programmeur – pardon, chaque fabricant de jouet – aura toute une panoplie de moyens pour mettre en place les permissions ou interdictions qu'il jugera utiles.

Cand. – Hm… Ne s’agit-il pas en fin de compte d’un nouveau modèle dogmatique qui restera en vogue jusqu’à ce qu’un nouveau ne le remplace dans quelques années ?

Doc. – Si on considère qu’il atteint son objectif, à savoir se rapprocher d’une organisation naturelle, on peut lui présager un futur à la hauteur !



Objectif objet : les aventures de l’OO

L’addition des méthodes dans la classe, dès que celles-ci portent sur un des attributs de la classe, transforme ces dernières de simples récipients d’information en véritables acteurs : l’objet fait plus qu’il n’est. Il ne se borne pas simplement à stocker son état ; il est surtout le premier responsable des modifications que celui-ci subit. Il sait et il fait, tout à la fois. Qu’un second objet, quelconque, désire se renseigner sur l’état du premier ou d’entreprendre de modifier cet état, il devra passer par les méthodes de ce premier objet qui, seules, ont la permission de lire ou transformer cet état. L’objet devra toujours être accompagné de son mode d’emploi que nous verrons plus loin, défini dans une structure de donnée à part : son interface.

L’orienté objet est loin d’être une pratique neuve en informatique, puisque le premier langage OO important, Simula, remonte à 1966. Cela permet aux vieux grisards de l’informatique, et qui cherchent à faire de leurs rides et de leurs cheveux blancs (on les attrape, paraît-il, beaucoup plus tôt en informatique), plus qu’une incitation au respect, un atout majeur, de prétendre que tout ce qui se fait d’apparemment neuf du côté de l’ordinateur n’est qu’un hoquet du passé, et que rien de vraiment novateur ne s’est produit depuis von Neuman ou Turing.

Kristen Nygaard et Ole-Johan Dahl : Simula

Simula est l’ancêtre de tous les langages orientés objet. Un ancêtre vieux seulement de 38 ans, car il a été proposé par deux chercheurs norvégiens, Kristen Nygaard et O-J. Dahl, en 1966, alors qu’ils étaient tous deux chercheurs au Norwegian Computing Center (NCC), à Oslo. Malgré son âge, il n’a pas pris une ride car tout y est de ce qui fait la force de l’OO : classe, objet, encapsulation, envoi de messages, typage fort, héritage, polymorphisme, multithreading, gestion mémoire, etc. Il fut à l’origine mis au point pour faciliter la conception et l’analyse des systèmes à temps discret, mais très vite évolua vers le langage de programmation fondateur de l’OO. Si le succès ne fut pas au rendez-vous, la raison en est simple : Simula était trop en avance sur son temps. Simula était la réponse logicielle la plus adéquate trouvée par ces chercheurs pour affronter la simulation de processus industriels complexes. La décomposition modulaire en classes suivait la décomposition structurelle du processus en ses différents composants. L’approche répondait à cette simple intuition : pourquoi baser la décomposition du logiciel sur un mode différent que celui qui vous est proposé par le monde ? Il semblait évident à ces deux chercheurs qu’écrire un programme c’est d’abord et avant tout réaliser un modèle de la réalité que l’on cherche à maîtriser à l’aide de celui-ci (d’où le nom de Simula).

Nygaard et Dahl furent extrêmement créatifs et productifs dans le département informatique de l’université d’Oslo et continuent à innover dans le développement des applications distribuées ou de langages OO plus compréhensibles. Nygaard est aussi très connu en Norvège comme un activiste politique et social des plus influents. Parallèlement à ses apports technologiques, il n’a pas cessé de se questionner sur l’impact des technologies de l’information dans la société et les systèmes d’éducation. Il fut très engagé dans les mouvements de protection de la nature et a été, surtout, le porte-étendard de la croisade qui enjoignit la Norvège à ne pas joindre l’Union européenne.

Ce n’est qu’assez récemment, en 2001 et 2002, que la communauté informatique a reconnu l’apport décisif de ces deux chercheurs dans l’informatique d’aujourd’hui, en leur décernant les prestigieux prix IEEE John Von Neuman et ACM Alan Turing, prix qui sont à l’informatique ce que le prix Nobel est aux autres sciences. Ces deux génies se sont suivis dans la créativité comme dans la mort. Ole-Johan Dahl nous a quittés le 29 juin 2002 à 70 ans, juste quelques semaines avant Kristen Nygaard parti, lui, le 9 août 2002 à 76 ans.

L’AITO, « Association Internationale pour les Technologies Objets » a, en 2004, créé le prix Dahl-Nygaard récompensant les informaticiens les plus importants dans l’évolution du monde OO. Il fut décerné en 2005 à Bertrand Meyer et en 2006 au « Gang des quatre », auteurs des Design Patterns présentés au chapitre 23.

Que les programmeurs en herbe se rassurent, on entend dire la même chose de la philosophie qui ne serait autre que des bas de page aux écrits de Platon, du jazz depuis Charlie Parker, et certainement de l’architecture, la peinture et de bien d’autres passe-temps humains. Vous aurez tôt fait de rétorquer à ces rabat-joie qu’il en va de même en matière de ringardise, où rien n’a plus vraiment évolué depuis les jérémiades du premier ringard…

Argumentation pour l’objet

Il est vrai que toutes les époques ne peuvent être aussi créatives, et que les années 1950 et 60 – époque charnière et témoin de la naissance de l’informatique – ont été inévitablement plus génératrices de nouveauté (il est plus commode d’innover à partir de rien). Néanmoins, il suffit de lorgner du côté de la bio-informatique ou de l’informatique quantique pour aisément se rendre compte que l’ordinateur est loin d’avoir épuisé ce potentiel de créativité qu’il suscite. Aujourd’hui, les progrès en matière de développement logiciel cherchent à rendre la programmation de plus en plus distante du fonctionnement intime des microprocesseurs, et de plus en plus proche de notre manière spontanée de poser et résoudre les problèmes.

Il est loin le temps où la maîtrise de l’assembleur était le signe distinctif de ceux qui savaient programmer. La simplicité de conception, l’accessibilité, l’adaptabilité, la fiabilité et la maintenance facilitée sont, bien plus que l’optimisation du programme en temps calcul et en espace mémoire, les voies du progrès. Ce qu’on perd en temps CPU, plusieurs indicateurs tendent à montrer qu’on le regagne aisément en espèces sonnantes et trébuchantes. Tout en informatique semble se conformer à la loi de Moore, d’un doublement de performance tous les 18 mois, tout … sauf les programmeurs. Plus l’application à réaliser est complexe et fait intervenir de multiples acteurs en interaction, plus il devient bénéfique de prendre ses distances par rapport aux contraintes imposées par le processeur, pour faire du monde qui nous entoure la principale source d’inspiration.

Transition vers l’objet

L’OO est une de ces étapes, qui ne demandent qu’à être poursuivies, à la croisée des progrès en génie logiciel, de l’amélioration des langages de programmation et des sciences cognitives. Mais foin de toute démagogie et de spéculation hasardeuse, et revenons à des considérations plus terriennes. Il est indéniable qu’il existe aujourd’hui deux manières de penser les développements logiciels : la manière dite « procédurale », et représentée par les langages de programmation de type procédural, comme C , Pascal, Fortran, Cobol, Basic, et la manière dite « orientée objet », et représentée par les langages de programmation de type orienté objet, comme C++, Java, Smalltalk, Eiffel, C#, Python.

Aujourd’hui, la seconde semble inéluctablement prendre le pas sur la première. Lors d’un entretien pour un emploi d’informaticien, répondez OO à toutes les questions, et vos chances d’embauche sont multipliées par 100. Une fois en place, programmez comme vous voulez et sans risque apparent car, alors que l’on peut mesurer votre degré d’alcoolémie, rien n’existe de semblable pour tester votre respect de la bonne pratique de l’OO. Hélas, dirons-nous, car à l’issue des prochains chapitres nous espérons que vous serez convaincus des nombreux avantages objectifs qu’il y a à adopter la pratique OO dans le développement d’applications logicielles un tant soit peu conséquentes.

De manière à différencier ces deux approches le plus pratiquement qui soit, nous nous glisserons dans la peau d’un biologiste qui désire réaliser la simulation d’un petit écosystème dans lequel, comme indiqué dans la figure 3-1, évoluent un prédateur (le lion), une proie (l’oiseau) et des ressources, eau et plante, nécessaires à la survie des deux animaux.

Mise en pratique

Simulation d'un écosystème

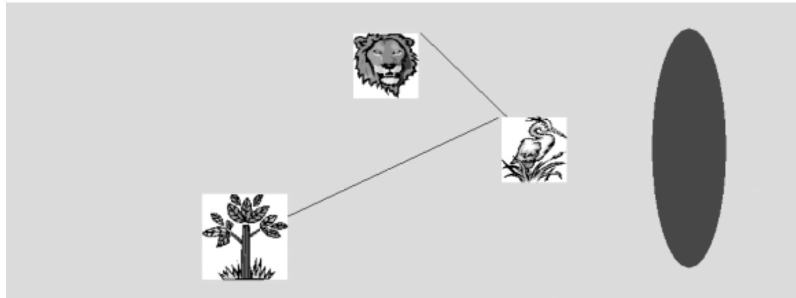


Figure 3-1
Programmation en Java d'un petit écosystème.

Décrivons brièvement le scénario de cette petite simulation. La proie se déplace vers l'eau, vers la plante, ou afin de fuir le prédateur ; elle agit en fonction du premier de ces objets qu'elle repère. La vision, tant de la proie que du prédateur, est indiquée par une ligne, qui part du coin supérieur de l'animal et balaie l'écran. Quand la ligne de vision traverse un objet, quel qu'il soit, l'objet est considéré comme repéré, la vision ne quitte plus l'objet, et l'animal se dirige vers la cible ou la fuit. Le prédateur se déplace vers l'eau ou poursuit la proie, là aussi en fonction du premier des deux objets perçus. L'énergie selon laquelle les deux animaux se déplacent décroît au fur et à mesure des déplacements, et conditionne leur vitesse de déplacement.

Dès que le prédateur rencontre la proie, il la mange. Dès que le prédateur ou la proie rencontre l'eau, ils se ressourcent (leur énergie augmente) et l'eau diminue de quantité (visuellement, la taille de la zone d'eau diminue). Dès que la proie rencontre la plante, elle se ressource (son énergie augmente également) et la plante diminue de quantité (sa taille diminue). Enfin, la plante pousse lentement avec le temps, alors que la zone d'eau, au contraire, s'évapore.

Analyse

Analyse procédurale

Adoptons dans un premier temps la pratique procédurale. Tous les objets de notre programme seront créés dès le départ, et stockés en mémoire des objets comme indiqué à la figure 3-2. Dans la phase d'élaboration structurelle des objets, rien ne distingue vraiment l'approche procédurale de l'approche OO. Les deux pratiques commencent à se démarquer par le fait que, dans la vision procédurale, les objets constituent un ensemble global de données du programme que de grands modules procéduraux affecteront tour à tour. En « procédural », l'analyse et la décomposition du programme se font de manière procédurale ou fonctionnelle, c'est-à-dire que l'on découpe le code en ses grandes fonctions.

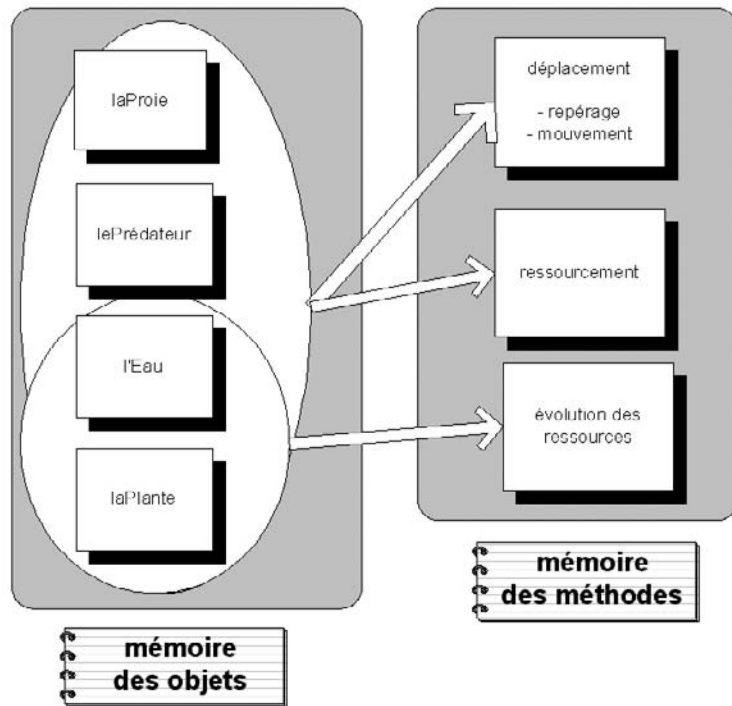
Programmation procédurale

La programmation procédurale s'effectue par un accès collectif et une manipulation globale des objets, dans quelques grands modules fonctionnels qui s'imbriquent mutuellement, là où la programmation orientée objet est confiée à un grand nombre d'objets, se passant successivement le relais, pour l'exécution des seules fonctions qui leur sont affectées.

Fonctions principales

Figure 3-2

Voici le découpage logiciel, avec ses grands blocs fonctionnels, de l'approche procédurale.



Quelles sont les fonctions que tous les objets doivent accomplir ici ? Tout d'abord, la proie et le prédateur doivent se déplacer. On commencera donc à penser et coder les déplacements de la proie et du prédateur, ensemble. Le fait de les traiter ensemble, même s'il s'agit de deux entités différentes, est ici très important. Tant pour le déplacement de la proie que du prédateur, il faudra préalablement que chacun des animaux repère une cible. La fonctionnalité de « déplacement » devra faire appel à une nouvelle fonctionnalité, de « repérage », qui, elle également, concerne tous les objets. En effet, les objets doivent se repérer entre eux : le prédateur cherche la proie, la proie cherche la plante et à éviter le prédateur, et tous deux cherchent l'eau. Le repérage se fait à l'aide de la vision, un bloc procédural constitué de deux fonctionnalités semblables, que l'on associera à chaque animal, et qui n'agira que pendant ce repérage.

Une deuxième grande fonctionnalité consiste dans le ressourcement de la proie et du prédateur. Ce ressourcement concernera à nouveau tous les objets car, pour la proie comme pour le prédateur, il fonctionne différemment selon la ressource rencontrée. Par exemple, lorsque la proie rencontre la plante, elle la mange et la plante s'en trouve diminuée. Lorsque le prédateur rencontre la proie, il la mange, et la proie, morte, disparaît de l'écran. La troisième et dernière fonctionnalité est l'évolution dans le temps des ressources, la plante poussant et l'eau s'évaporant. Le programme principal se trouvera finalement constitué de tous les objets du problème, et ensuite de trois grandes procédures : « déplacement », « ressourcement » et « évolution des ressources ». La première d'entre elles fait appel à une nouvelle procédure de repérage entre les objets. Cette décomposition fonctionnelle est représentée dans la figure 3-2.

Conception

Conception procédurale

Le déplacement porte sur tous les objets, le repérage les confronte deux à deux. Le ressourcement porte également sur tous les objets. L’évolution des ressources ne porte que sur deux des objets. On comprend par cet exemple comment la pratique procédurale découpe l’analyse du problème en de grandes fonctions, imbriquées, et portant, toutes, sur l’essentiel des données du problème. En « procédural », les grandes fonctions accomplies par le logiciel, en s’imbriquant l’une dans l’autre, sont la voie de la modularité et de l’évolution de toute l’approche. On perçoit aisément, tant par le partage collectif des objets que par l’interpénétration des fonctions, qu’il est plus difficile de parvenir à une décomposition naturelle du problème en des modules relativement indépendants.

Conception objet

Pour sa part, la pratique orientée objet cherche d’abord à identifier les acteurs du problème et à les transformer en classe, regroupant leurs caractéristiques structurelles et comportementales. Les acteurs ne se limitent pas à exister structurellement, ils se remarquent surtout par le comportement adopté, par ce qu’ils font, pour eux et pour les autres. Ce ne sont plus les grandes fonctions qui guident la construction modulaire du logiciel, mais bien les classes/acteurs eux-mêmes. Les acteurs ici sautent aux yeux. Il s’agira de la proie, du prédateur, de l’eau et de la plante. Une fois que l’on a établi les attributs de chacun, la grande différence avec la démarche précédente consistera à réaliser une nouvelle analyse fonctionnelle, mais particularisée à chaque acteur, cette fois.

Que font, pris individuellement, la proie, le prédateur, l’eau et la plante ? Commençons par les plus simples. La plante pousse et peut diminuer sa quantité, l’eau s’évapore et peut également diminuer sa quantité. La proie peut se déplacer, et doit pour cela repérer les autres objets. À cette fin, elle utilisera, comme le prédateur, une nouvelle classe `vision`, constituée d’une longueur, et à même de repérer quelque chose dans son champ. Mais la proie se devra d’interagir avec les autres classes. La proie peut boire l’eau et manger la plante. L’interaction avec les autres classes se poursuit. Le prédateur, à son tour, se déplacera en fonction des cibles, et peut boire l’eau et manger la proie. Les liens entre objets et méthodes sont maintenant représentés comme dans la figure 3-3.

Impacts de l’orientation objet

Les acteurs du scénario

Ici, le découpage logiciel s’effectue à partir des grands acteurs de la situation, et non plus des grandes activités propres à la situation. Un avantage évident est que le premier découpage apparaît bien plus naturel à mettre en œuvre que le second et, comme vous le savez sans doute, si vous chassez le naturel, il revient à l’OO. Il est plus intuitif de séparer le programme à réaliser en ces quatre acteurs qu’en ses trois grandes activités. Vous ne percevez pas la jungle comme la réunion de trois grandes activités : migratoire, évolutive et alimentaire, vous la percevez, d’abord et avant tout, comme un regroupement d’animaux et de végétaux.

Un autre avantage, que nous étaierons par la suite, est que, dans l’approche procédurale, toutes les activités impliquent tous les objets. Cela a pour effet d’accroître les difficultés de maintenance et de mise à jour du logiciel. Changez quoi que ce soit dans la description d’un objet, et vous risquez d’avoir à ré-éplucher l’entièreté du code. Au contraire, dans l’approche OO, si vous changez la description structurelle de l’eau, au pire, c’est la seule méthode `évaporer` qu’il faudra ré-examiner.

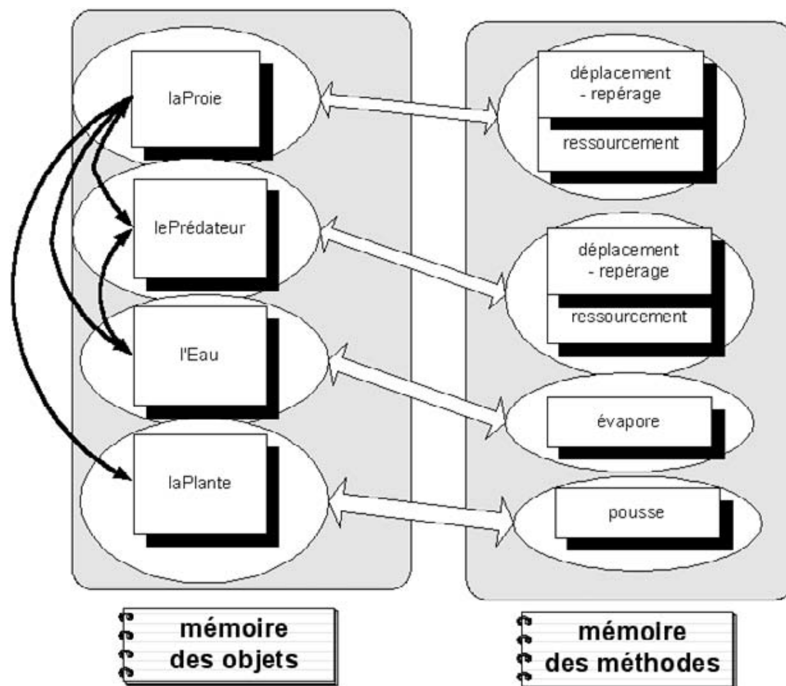


Figure 3-3

Dans l'approche OO, le découpage du logiciel se fait entre les classes qui regroupent les descriptions structurelles avec les activités les concernant. Les classes interagissent entre elles.

Indépendance de développement et dépendance fonctionnelle

Le renforcement de l'indépendance entre les classes est une volonté majeure de la programmation OO, et nous reviendrons largement sur ce point dans les prochains chapitres. Cependant, il est important de distinguer d'emblée l'indépendance dans le développement logiciel des classes de la dépendance fonctionnelle entre ces mêmes classes, et qui reste la base de l'OO. Nous verrons dans les prochains chapitres qu'en encapsulant les classes, on favorise leur développement de manière indépendante bien que, lors de leur passage à l'action, ces classes soient fonctionnellement dépendantes.

Dépendance fonctionnelle *versus* indépendance logicielle

Alors que l'exécution d'un programme OO repose pour l'essentiel sur un jeu d'interaction entre classes dépendantes, tout est syntaxiquement mis en œuvre lors du développement logiciel pour maintenir une grande indépendance entre les classes. Cette indépendance au cours du développement favorise tant la répartition des tâches entre les programmeurs que la stabilité des codes durant leur maintenance, leur réexploitation dans des contextes différents et leur évolution.

Ainsi, la dépendance entre les classes, quand dépendance fonctionnelle il y a, par exemple ici, entre la classe prédateur et la classe proie, se réalise directement entre les deux classes en question, sans un détour obligé par un module logiciel, plus global, les regroupant en son sein. Le prédateur devra repérer la proie, puis la rattraper pour finalement la dévorer. Il le fera en activant un certain nombre de ses méthodes, qui, de leur côté, nécessiteront de lancer l'exécution de méthodes directement sur la proie. De manière semblable,

lorsque la proie boira l’eau, elle activera pour ce faire la méthode de la classe `Eau`, responsable de sa diminution de quantité.

En substance, les dépendances entre classes se pensent au coup par coup et deux à deux, et ne sont pas noyées dans la mise en commun de toutes les classes dans les modules d’activité logicielle. Cela conduit à une conception de la programmation sous la forme d’un ensemble de couples client-fournisseur (ou serveur), dans laquelle toute classe sera tour à tour client et fournisseur d’une ou de plusieurs autres. Une conception où les classes se rendent mutuellement des services, ou se délèguent mutuellement des responsabilités, pointe à l’horizon. Une conception bien plus modulaire que la précédente, car le monde contient plus d’acteurs que de fonctions possibles. Les grandes fonctions génériques sont redéfinies pour chaque acteur.

Petite allusion (anticipée) à l’héritage

Ce dernier point conduira très naturellement à la pratique de l’héritage et du polymorphisme, pratique dont la non-invocation ici, pour la programmation du petit écosystème, nous amène à différer de quelques chapitres sa modélisation complète en orienté objet (nous la reprendrons au chapitre 11). En effet, si vous êtes déjà passés sur les fonts baptismaux de l’OO, le fait que cette simulation soit propice à une mise en pratique des mécanismes d’héritage ne vous aura pas échappé. Du moins nous l’espérons, sinon, point de regret, ce livre était un investissement nécessaire. Rassurez-vous, cela ne nous a pas échappé non plus et, dans un prochain chapitre, nous reviendrons sur cet écosystème, en multipliant les objets qui le constituent, mais surtout en rajoutant quelques superclasses du côté des animaux et des ressources.

La collaboration des classes deux à deux

Un ensemble de classes, agissant par paire et par envoi de messages, ici encore, cela permet de favoriser l’éclatement du programme, en forçant autant que faire se peut les classes à devenir indépendantes entre elles, car, deux par deux, c’est toujours mieux que toutes avec toutes. Plus le programme est décomposable, plus facile sera l’attribution de ses modules à différents programmeurs, et plus réduit sera l’impact provoqué par le changement d’un de ses modules sur les autres. Quoi de plus naturel, dès lors, que de décomposer un logiciel, en collant au mieux à la manière dont notre appareil perceptif et nos facultés cognitives découpent le monde. Une perception qui, pour l’essentiel, sépare les objets, et qui, pendant quelques minutes (au parloir), leur permet de s’échanger quelques messages.

OO comparé au procédural en performance et temps calcul

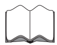
Il est parfaitement incorrect de clamer haut et fort que les performances en consommation des ressources informatiques (temps calcul et mémoire) des programmes OO sont supérieures en général à celles de programmes procéduraux remplissant les mêmes tâches. Tout concourt à faire des programmes OO de grands consommateurs de mémoire (prolifération des objets, distribués n’importe où dans la mémoire violant les principes de « localité » informatique) et de temps calcul (retrouver à l’exécution les objets et les méthodes dans la mémoire, puis traduire, au dernier moment, les méthodes dans leur forme exécutable sans parler du garbage collector et autres « casseur » de performance). Les seuls temps et ressources réellement épargnés sont ceux des programmeurs, tant lors du développement que lors de la maintenance et de l’évolution de leur code. L’OO considère simplement, à juste titre nous semble-t-il, que le temps programmeur est plus précieux que le temps machine.

4

Ici Londres : les objets parlent aux objets

Ce chapitre illustre et décrit le mécanisme d'envoi de messages qui est à la base de l'interaction entre les objets. Cette interaction exige que les classes dont sont issus ces objets entrent dans un rapport de composition, d'association ou de dépendance. Ces messages peuvent s'enclencher de manière récursive.

Sommaire : Envoi de messages — Composition, association et dépendance entre classes — De la sévérité du compilateur — Réaction en chaîne d'envois de messages



Candidus — Alors, y'en a plus que pour les objets, les procédures passent donc à la trappe, c'est bien ça ?

Doctus — Pas tout à fait. Chaque objet a ses voyants et ses boutons : les voyants affichent la valeur des données à chaque instant ; les boutons actionnent ses méthodes. Ces méthodes ne sont pas autre chose que nos anciennes procédures.

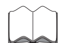
Cand. — Ainsi, la nourrice incite notre bébé à activer les bons boutons et c'est lui qui déclenche la suite des événements ?

Doc. — Le déclenchement des méthodes d'un objet est effectivement conditionné par la disponibilité des boutons de commande correspondants. Mais nous aurons également affaire à quelques mécanismes plus subtils. Les jouets les plus complexes contiendront des mécanismes internes que le fabricant du jouet n'a pas mis à portée de main. Ils ne concernent que le fonctionnement intime de notre objet. Par ailleurs, certains de nos objets peuvent être combinés de telle sorte que le fonctionnement de l'un en mette un ou plusieurs autres en action.

Ig — On aurait donc aussi des interfaces apparentes pour les connexions entre objets. Ne deviennent-ils pas complètement dépendants les uns des autres ?

Doc. — Oui, on aboutit à un circuit de dépendance. Ce qui amène la question suivante : comment allons-nous nous assurer du bon cloisonnement entre les différents objets sans nous interdire de les combiner quand c'est possible ?

Cand. — Hm… Tu voudrais faire des usines à gaz sans trop de tuyaux quoi !



Envois de messages

David A. Taylor : *Object Technology: A Manager's Guide* (Addison-Wesley)

Cet ouvrage existe en français sous le titre *Technologie orienté objet* chez Vuibert. C'est une excellente introduction à l'orienté objet, qui réussit sans aucun approfondissement technique, à communiquer tout l'esprit de la conception et de la programmation OO. L'ouvrage est agrémenté d'un ensemble de dessins originaux, spirituels et surtout didactiques. La biologie et les systèmes complexes y sont largement à l'honneur (ce qui, cela va sans dire, n'est pas pour nous déplaire). C'est notre porte d'entrée favorite dans le monde de l'OO, construite par un auteur enthousiaste, cultivé, s'adressant aux néophytes et à tous ceux qui, bien que concernés par l'OO, ne mettront peut-être jamais les mains dans du code. Non seulement les mécanismes clefs de l'OO y sont abordés, mais l'auteur s'attaque avec la même superficialité brillante à la problématique des objets distribués et de leur stockage dans les bases de données objet ou objet-relationnelle (voir chapitre 19).

Dans les chapitres précédents, nous avons discuté de la nécessité de faire interagir l'objet `feu de signalisation` avec l'objet `voiture`, quand le passage au vert du premier déclenche le démarrage du second. De même, nous avons retrouvé un type semblable d'interaction quand le lion, s'abreuvant, provoque la diminution de la quantité d'eau. Le lion ne peut directement diminuer la quantité d'eau, quelle que soit l'ampleur de sa soif, qu'il meure ou non ce soir. Il le fera par un appel indirect à la méthode, responsable pour l'eau, de la diminution de sa quantité. L'eau gère sa quantité, pas le lion, qui, lui, a déjà fort à faire avec la gestion de son énergie et de ses déplacements.

Les objets interagissent, et comme tout ce qu'ils font doit être prévu dans leurs classes, celles-ci se doivent d'interagir également. C'est cette interaction entre objets, lorsqu'un d'entre eux demande à l'autre d'exécuter une méthode, qui constitue le mécanisme clé et récurrent de l'exécution d'un programme OO. Un tel programme n'est finalement qu'une longue et barbante litanie d'envois de messages entre les objets agrémentés ici et là de quelques mécanismes procéduraux (tests conditionnels, boucles…).

Nous allons, à l'aide d'un exemple minimal de programmation, illustrer ce principe de communication entre deux objets. Supposons un premier objet `o1`, instance d'une classe `O1`, contenant la méthode `jeTravaillePourO1()`, et un deuxième objet `o2`, instance d'une classe `O2`, contenant la méthode `jeTravaillePourO2()`. Dans le logiciel, `o1` interagira avec `o2`, si la méthode `jeTravaillePourO1()` contient une instruction comme: `o2.jeTravaillePourO2()`. C'est au moment précis de l'exécution de cette instruction que l'objet `o1` interférera avec l'objet `o2`. Mais, pour que `o1` puisse exécuter quoi que ce soit, il faudra d'abord déclencher la méthode `jeTravaillePourO1()`sur `o1`. Nous supposerons que la méthode `main` s'en charge et débute l'exécution du programme en déclenchant la méthode `jeTravaillePourO1()` sur l'objet `o1`. Le reste suivra, comme indiqué à la figure 4-1.

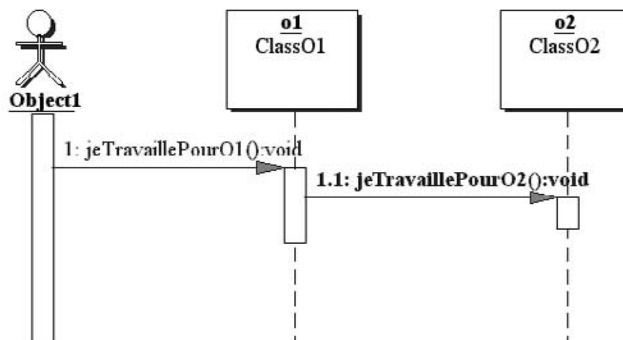


Figure 4-1
L'objet o1 parle à l'objet o2.

Dans ce diagramme (il s'agit en fait d'un diagramme de séquence UML que nous préciserons au chapitre 10 mais nous pensons qu'il est compréhensible en l'état), le petit bonhomme fait office de `main`, en envoyant le premier message `jeTravaillePourO1()` sur l'objet `o1`. Par la suite, nous voyons que l'objet `o1` interrompt l'exécution de sa méthode, le temps pour `o2` d'exécuter la sienne. Finalement, le programme reprendra normalement son cours, là où il l'a abandonné, en redonnant la main à `o1`. L'envoi de message s'accompagne d'un passage de relais de l'objet `o1` à l'objet `o2`, relais qui sera restitué à `o1` une fois qu'`o2` en aura terminé avec sa méthode.

Association de classes

Nous avons déjà rencontré ce chien de garde sévère qu'est le compilateur, et qui ne laisse rien passer, si ce que font les objets n'a pas été prévu dans leur classe. Ainsi, si `o1` déclenche la méthode `jeTravaillePourO2()` sur l'objet `o2`, c'est que la classe responsable de `o1` sait que cet objet `o2` est à même de pouvoir exécuter cette méthode.

Figure 4-2
Les deux classes O1 et O2 en interaction par un lien fort et permanent dit « d'association ». Les messages sont envoyés de la classe `O1` vers la classe `O2`.

Pour ce faire, la classe `O1` se doit d'être informée, quelque part dans son code, sur le type de l'objet `o2`, c'est-à-dire la classe `O2`. Une première manière pour la classe `O1` de connaître la classe `O2` est celle indiquée par la figure 4-2 (il s'agit à nouveau d'un petit diagramme UML, cette fois de classe, que nous préciserons aussi au chapitre 10, mais nous pensons là encore qu'il est très compréhensible en l'état). On dira dans ce cas que les deux classes sont associées, et que la connaissance de `O2` devient une donnée structurelle à part entière de la classe `O1`. En langage de programmation, `O2` devient purement et simplement le type d'un attribut de `O1`, comme dans le petit code de la classe `O1` qui suit :

```
class O1 {
  O2  lienO2 ; /*la classe O2 type un attribut de la classe O1 */
  void  jeTravaillePourO1() {
    lienO2.jeTravaillePourO2() ;
  }
}
```

Cela peut être le cas si les deux objets qui interagissent entrent dans une relation de composition. Si telle est leur manière d'être ensemble, chaque objet de la classe `O1` contiendra « physiquement » un objet de la classe `O2`, et ces deux objets deviendront alors indéfectiblement liés dans la vie comme dans la mort. Mais cela peut être plus simplement le cas si `o1` désire pouvoir, partout dans son code (à l'intérieur de toutes ses méthodes), envoyer des messages à `o2`. Comme tout autre attribut de type « primitif » (entier, réel, booléens…), cet attribut `lienO2` devient accessible dans l'entièreté de la classe. En cela, on peut dire que cet attribut d'un type un peu particulier fait de la liaison entre les deux classes une vraie propriété structurelle de la première. Il s'agit en fait d'une espèce nouvelle d'attribut dit « attribut référent », typé, non plus par un type primitif, mais bien par la classe qu'il réfère. Dans l'espace mémoire réservé à `o1`, l'attribut `lienO2` contiendra, comme indiqué dans la figure 4-3, l'adresse de l'objet `o2`.

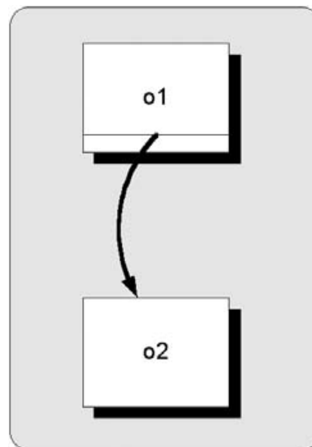


Figure 4-3
Comment o1 possède l'adresse de o2.

Cela permettra à `o1` de connaître parfaitement l'adresse de son destinataire, ce qui est souhaitable lorsqu'on désire lui faire parvenir un message. Cette notion de message prend tout son sens en présence de ces deux acteurs essentiels que sont l'objet expéditeur et l'objet destinataire. Comme conséquence d'une association dirigée entre la classe `O1` et la classe `O2`, tout objet de type `O1` sera lié à un objet de type `O2`, avec lequel il pourra communiquer à loisir, indépendamment de son état, et quelles que soient les activités entreprises. Le compilateur, en compilant O1, devra être informé de l'emplacement physique d'O2, tout comme à l'exécution.

Association de classes

La manière privilégiée pour permettre à deux classes de communiquer par envoi de message consiste à rajouter aux attributs primitifs de la première un attribut référent du type de la seconde. La classe peut donc, tout à la fois, contenir des attributs et se constituer en nouveau type d'attribut. C'est grâce à ce mécanisme de typage particulier que, partout dans son code, la première classe pourra faire appel aux méthodes disponibles de la seconde.

Dépendance de classes

Mais il existe d'autres manières, non persistantes cette fois, pour la classe `O1` de connaître le type `O2`. Comme dans le code qui suit, la méthode `jeTravaillePourO1(O2 lienO2)` pourrait recevoir comme argument l'objet `o2`, sur lequel il est possible de déclencher la méthode `jeTravaillePourO2()`.

```
class O1 {
  void  jeTravaillePourO1(O2 lienO2) {
       lienO2.jeTravaillePourO2() ;
  }
}
```

Le compilateur acceptera le message car le destinataire est bien typé par la classe `O2`. Cependant, dans un cas semblable, le lien entre `O1` et `O2` n'aura d'existence que le temps d'exécution de la méthode

`jeTravaillePour01()`, et on parlera, entre les deux classes, plutôt que d'un lien d'association, d'un lien de dépendance, plus faible et surtout passager (voir figure 4-4).

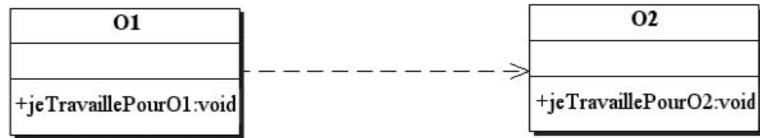


Figure 4-4
Les deux classes O1 et O2 en interaction par un lien faible et temporaire dit de « dépendance ».

La raison en est que, lors de l'appel de toute méthode, se crée un espace de mémoire temporaire, dans lequel sont stockés les arguments transmis à la méthode. Il s'agit en fait d'un type de mémoire dénommée « pile », associée à la méthode, disparaissant avec elle, et sur laquelle nous aurons l'occasion de revenir. À la fin de l'exécution de la méthode, cet espace est perdu et restitué au programme, en conséquence de quoi le lien entre les deux objets s'interrompt immédiatement par disparition du référent temporaire. Les deux classes ne se connaissent dès lors que durant le temps d'exécution de la méthode. Un lien de dépendance est maintenu entre les deux classes, car toute modification de la classe qui fournit la méthode à utiliser pourrait entraîner une modification de la classe qui fait appel à cette méthode. Il s'agit à proprement parler plutôt d'une dépendance de type logicielle, car elle reste au niveau de l'écriture logicielle des deux classes, alors que le lien d'association va bien au-delà de cette dépendance, et représente une véritable connexion structurelle et fonctionnelle entre ces deux classes.

Une autre liaison passagère pourrait se produire, si, comme dans le code ci-après, la méthode `jeTravaille-Pour01()` décide, tout de go, de créer l'objet `lien02`, le temps de l'envoi du message.

```
class O1 {
  void  jeTravaillePour01() {
    O2 lien02 = new O2();
    lien02.jeTravaillePour02() ;
  }
}
```

Au sortir de la méthode, l'objet `lien02` sera irrémédiablement perdu, de même que cette liaison passagère qui, là encore, n'aura duré que le temps d'exécution de la méthode `jeTravaillePour01()`. Le seul référent étant à nouveau stocké en mémoire pile, la méthode terminée, ce référent disparaîtra emportant à sa suite le seul objet référé par lui. Alors que, dans la première dépendance, c'est la liaison entre les deux objets qui s'interrompait à la fin de la méthode, ici l'objet destinataire du message s'éclipsera également à la fin de cette méthode. Autant que faire se peut, on privilégiera entre les classes des liens de type fort, d'association, faisant des relations entre les classes une donnée structurelle de chacune d'entre elles. Il vaut mieux, dès lors qu'elles envisagent la moindre communication, que les classes se connaissent, non pas intimement, comme nous le verrons, mais suffisamment, pour échanger quelques bribes de conversation tout au long de leur existence.

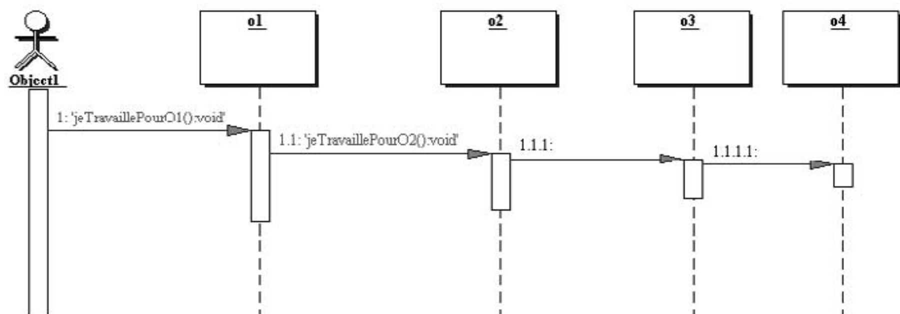
Les fichiers contenant les classes en interaction, et pour autant qu'on informe les phases de compilation et d'exécution de l'emplacement de ces fichiers, s'en trouveront automatiquement liés également. De même, ce lien structurel entre les classes et leurs instances sera préservé lorsque ces objets seront sauvegardés de manière permanente. Y compris sur le disque dur, les objets posséderont parmi leurs attributs des référents connaissant l'adresse physique sur le disque des objets avec lesquels ils sont en relation (comme nous le verrons plus en détail au chapitre 19).

> **Communication possible entre objets**
> Deux objets pourront communiquer si les deux classes correspondantes possèdent entre elles une liaison de type composition, association ou dépendance, la force et la durée de la liaison allant décroissant avec le type de liaison. La communication sera dans les deux premiers cas possible, quelle que soit l’activité entreprise par le premier objet, alors que dans le troisième cas elle se déroulera uniquement durant l’exécution des seules méthodes du premier objet, qui recevront de façon temporaire un référent du second.

Réaction en chaîne de messages

Finalement, il sera très fréquent que l’objet `o2` lui-même, durant l’exécution de sa méthode `jeTravaillePourO2()`, envoie un message vers un objet `o3`, qui lui-même enverra un message vers un objet `o4`, etc. On assiste donc, comme dans la figure 4-5, à une succession d’envois de messages en cascade. Toutes les méthodes successivement déclenchées seront, dans une approche séquentielle traditionnelle (où seule une méthode à la fois peut requérir le processeur – nous évoquerons d’autres approches, où plusieurs messages peuvent s’exécuter en parallèle, au chapitre 17), imbriquées les unes dans les autres. Le flux d’exécution passera de l’une à l’autre pour, à la fin de l’exécution de la dernière, refaire le chemin dans le sens inverse, et achever successivement toutes les méthodes enclenchées.

Figure 4-5
Un envoi de messages en cascade.



Bertrand Meyer compare cela au déploiement successif de toutes les pièces d’un feu d’artifice géant et complexe, résultant directement ou indirectement de l’allumage initial d’une minuscule étincelle. Dans le chapitre 10, nous racontons comment la réalisation et les conséquences d’un but lors d’une simulation orientée objet d’un match de football peut occasionner la participation et l’interaction de nombreux objets de classe différente : Ballon, Joueurs, Arbitre, Filets, Score, Terrain… sans pour cela que l’écriture du code ne s’en trouve inutilement compliquée.

> **Réaction en chaîne**
> Tout processus d’exécution OO consiste essentiellement en une succession d’envois de messages en cascade, d’objet en objet, messages qui, selon le degré de parallélisme mis en œuvre, seront plus ou moins imbriqués.

Exercices

Exercice 4.1

Considérez les deux classes suivantes : `Interrupteur` et `Lampe`, telles que, quand l'interrupteur est allumé, la lampe s'allume aussitôt. Réalisez dans un pseudo-code objet le petit programme permettant cette interaction.

Exercice 4.2

Tentez d'envisager dans quelles circonstances il est préférable de privilégier entre deux classes en interaction une relation de type « dépendance » plutôt qu'une relation de type « association ».

Exercice 4.3

Réalisez un petit programme exécutant l'envoi de message entre deux objets, et ce lorsque les deux classes dont ils sont issus entretiennent entre elles une relation de type « association » dans le premier cas, et de « dépendance » dans le second.

Exercice 4.4

Écrivez un squelette de code réalisant un envoi de messages en cascade entre trois objets.

Exercice 4.5

Considérez les deux classes suivantes : Souris et Fenêtre, telles que, quand la souris clique sur un point précis de la fenêtre, celle-ci se ferme. Réalisez un pseudo-code objet permettant cette interaction. On favorisera une relation de type dépendance entre les objets concernés.

5

Collaboration entre classes

Le but de ce chapitre est de poursuivre la découpe d'une application OO en ses classes, classes jouant, tout à la fois, le rôle de module, fichier et type. Java favorise de façon très pratique cette vision, étendant la relation qui existe entre les classes jusqu'aux étapes de compilation et d'exécution. C++, Python, PHP 5 et C# rendent cette mise en œuvre moins immédiate, les trois premiers en raison d'un souci de compatibilité avec le monde procédural y compris le C, le dernier avec Windows et les exécutables d'avant.

Sommaire : Les classes comme fichiers reliés — La compilation dynamique — Association de classes — Auto-association de classes — Les paquetages

Candidus — J'aimerais bien faire une visite chez le fabricant de jouets, histoire de voir comment il se débrouille avec tout ça, comment il construit puis emballe ce qu'il livre à notre bébé…

Doctus — L'organisation des fichiers en différentes catégories permet de rechercher les objets comme dans un jeu de pistes. Ils auront le même nom que l'objet qu'ils contiennent. Bébé n'aura alors qu'à utiliser des règles simples pour trouver tout ce qui lui sera nécessaire. Si le fichier contenant l'objet ne se trouve pas là où il devrait être, il ne sera pas content et nous le fera savoir !

Cand. — Ne risque-t-on pas d'avoir des conflits avec des noms d'objets homonymes ?

Doc. — Bien sûr que si, c'est pourquoi on ne peut pas avoir plusieurs fichiers portant le même nom dans un même répertoire ! La solution consiste à utiliser les packages, pour que chaque fabricant ait un répertoire d'entrée différent… et le tour est joué !

Candidus — Bébé utilise donc les mêmes règles que le fabricant pour déduire la position des objets. Lorsqu'une pièce du puzzle en appelle une autre, par son nom, il n'y aura qu'à suivre la piste indiquée pour mettre la main dessus.

Doc. — Oui, et par-dessus le marché, ces règles simples font que le fabricant peut s'assurer que tous les liens entre les pièces de ses jouets figurent bien dans sa livraison.

Cand. — L'informatique… mais c'est très simple !

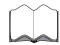
Doc. — Mieux encore : même les moules qu'utilisent les fabricants pour créer les objets sont organisés ainsi. Ces derniers seront tout simplement déployés dans une structure de répertoires identique, les noms de fichiers ne seront distingués que par un suffixe différent pour le moule et l'objet fini.

Cand. — Tous ces liens peuvent nous faire un sacré labyrinthe si un grand nombre d'objets se connaissent l'un l'autre. Lors de la fabrication, si un objet est combiné à un autre qui n'est pas encore sorti de son moule, comment le fabricant s'y prend-il ? Il lui faudra manipuler toute une série de moules en même temps avant d'en avoir terminé avec cet objet composite. Ça semble bien compliqué tout ça !

Doc. — Il faut tout simplement se rappeler la chose suivante : pour savoir utiliser un objet accessoire, peu importe qu'il soit ou non immédiatement disponible si son mode d'emploi est bien défini. Il nous suffira qu'il soit effectivement livré quand bébé en aura besoin pour faire fonctionner son puzzle animé.

Cand. — Ce qui fait qu'un objet peut avoir une existence virtuelle avant d'être vraiment réalisé – ça ressemble à l'histoire de la poule et de l'œuf !

Doc. — Exactement, mais ce n'est plus un problème dans notre cas. Il s'agit simplement de savoir en quoi consistera chaque objet et ce qu'on pourra en attendre pour connaître entièrement ce qu'on doit en savoir. Ce n'est qu'au moment de l'emballage qu'il s'agira de vérifier que tous les liens s'emboîteront correctement.



Pour en finir avec la lutte des classes

Nous avons vu dans les chapitres précédents que ce qu'il y a sans doute de plus capital en orienté objet, le Capital, c'est d'en finir avec la lutte des classes. Les classes ne sont pas de simples structures d'accueil d'information, des « inforooms », elles sont à notre service pour la réalisation de certaines tâches mais, plus encore, elles sont chacune au service des autres. Elles le sont, car elles ne peuvent déléguer à aucune autre la responsabilité de l'évolution de leur état. Si les autres nécessitent une modification de l'état d'une classe, elles doivent impérativement s'adresser à elle.

On ne le répétera jamais assez, la programmation orientée objet se conçoit, essentiellement, comme une société de classes en interaction, se déléguant mutuellement un ensemble de services. Les classes, lors de leur conception, prévoient ces services, pour que le compilateur s'assure de la cohérence et de la qualité de cette conception, et traduise le tout en une forme exécutable. Par la suite, ces services s'exécuteront en cascade, quand les objets, instances de ces classes, occuperont la mémoire et se référenceront mutuellement, afin de réaliser le programme anticipé par la structure relationnelle des classes.

Java , James Gosling et Bill Joy

James Gosling, d'origine canadienne et aujourd'hui directeur technologique chez Sun Microsystems, aurait été bien incapable, il y a de cela une douzaine d'années, de pressentir le succès extraordinaire que rencontrerait le langage de programmation sur lequel il travaillait. Le projet « Green », qui donnera naissance à ce langage, langage appelé « Oak » à l'origine, avait pour finalité la programmation de petits appareils électriques et électroniques de grande consommation, comme de l'électroménager, télévision, chaînes hifi, et autres, tous dotés de processeurs de conception différente. Il fallait un langage simple, sûr, portable. La portabilité fut empruntée au langage Pascal, pour lequel Niklaus Wirth avait déjà, à l'époque, imaginé le principe de la machine virtuelle, et conçu celle adaptée au Pascal. Il fallait un langage d'utilisation simple et intuitive, l'OO s'imposait mais, préservant la culture Unix chère à tous les informaticiens, la syntaxe C/ C++ s'imposait. Le projet Green ne rencontra pas le succès escompté, et Java aurait pu sombrer dans les oubliettes des créations informatiques sans le succès soudain et explosif du Web.

Le Web était en 1995 uniquement statique, une page web se bornant à apparaître sans aucune possibilité d'interaction. Le navigateur « Mosaic » commençait à largement se répandre, mais sans solutionner l'austère inertie des pages web. Gosling raconte que, lors d'une conférence dédiée à Internet, il présenta un nouveau concept de navigateur sur lequel il travaillait. Il fit d'abord apparaître une molécule dans une page web classique. L'assistance resta de marbre. Mais quelle ne fut la surprise de ce même public lorsque, à l'aide de la souris, il se mit à bouger la molécule, la faisant tourner, la déplaçant en avant et en arrière. Une applet Java s'exécutait dans le navigateur, qui permettait cela.

L'engouement pour Java démarra ce même jour J (comme Java). L'avantage essentiel de Java dans le monde d'Internet, un monde informatiquement hétérogène (n'importe quelle plate-forme informatique pouvant devenir un nœud du réseau), était sa capacité à s'exécuter d'une seule et même manière quel que soit le couple processeur/OS sur lequel l'applet s'exécutait. La décision de Netscape de rendre la version 2.0 compatible avec Java (d'y intégrer une machine virtuelle Java) ne fit qu'accroître son succès. On connaît la suite.

Il est étonnant de voir que le succès de ce langage tient au début, non pas à ses qualités syntaxiques propres, mais en ce qu'il propose une solution technique pour rendre les pages Web plus dynamiques et interactives. Depuis, bien d'autres technologies permettent aux pages Web d'exécuter du code en local, telles que VBScript, JavaScript. Java reste pourtant le langage privilégié des applications Internet, de par l'exploitation de sa technologie RMI (voir chapitre 16), dont les objets bénéficient pour se rendre mutuellement des services à travers le Web. Aujourd'hui, force est de constater que le succès de Java a largement dépassé ce premier cadre d'application. Il est devenu, tout comme pour nous ici, le langage idéal pour l'enseignement de l'OO, au détriment du C++, dont la complexité est par trop rébarbative pour une première entrée dans le monde de l'OO.

Il s'impose d'ailleurs comme tel dans un nombre croissant de centres d'enseignement de l'informatique (plus de la moitié, dit-on, à ce jour). Il est aussi le langage de programmation le plus utilisé et semble le plus apprécié lorsque des sondages sont effectués auprès des informaticiens.

Toutes les qualités reconnues de ce langage et reprises dans le livre blanc de Java : langage simple, OO, portable, distribué, fiable, performant, multithread, dynamique, étaient au départ destinées à son exploitation dans un appareillage varié, fragile, aux capacités informatiques modestes et dont il fallait préserver une grande facilité d'utilisation. Ce sont ces mêmes qualités qui ont fait de ce langage un moyen idéal d'assimiler les principes de l'OO. Même les plus ardents défenseurs du C++, à commencer par Bjarne Stroustrup lui-même (et qui a toujours su qu'il existait dans C++ un petit noyau syntaxique plus compact et plus cohérent qui ne demandait qu'à germer), ont salué avec enthousiasme l'élégance et les qualités de ce langage.

Gosling a vu dans la luxuriance syntaxique du C++ et les multiples degrés de liberté qui lui sont inhérents une source d'erreurs et de maladresses. On comprend, vu la cible première, non plus les fiers ordinateurs, mais des appareils ménagers, qu'il ait préféré délester le programmeur d'une part de son contrôle lors de l'exécution du programme (ainsi l'absence de pointeurs explicites et de gestion mémoire manuelle). Si son pari initial, « Green », ne fut pas gagné à l'époque, des projets comme « Jini » tentent de le repositionner en ligne de mire. En effet, qu'est-ce que Jini (évoqué chapitre 19) si ce n'est le développement d'une plate-forme de programmation pour tout type de processeur embarqué dans tout type d'appareil connecté à Internet, ordinateur, caméra, radio ou frigo. Gossling travaille de manière intensive sur l'interconnectivité de ces différents types d'appareils : vous filmez et ce que vous filmez apparaît derechef sur le bac à glaçons de votre réfrigérateur ou sur l'écran de votre télévision. Vous conduisez et êtes informé en ligne de tous les problèmes de trafic via Internet. Il s'investit aussi dans de nouvelles manières de programmer, plus visuelles, et donnant plus d'importance à des représentations graphiques, de type arborescentes, de la structure et du fonctionnement des programmes.

Une petite mention supplémentaire pour Bill Joy, co-fondateur, à l'âge de 28 ans, de Sun Microsystems avec Scott McNealy (son patron actuel) et un des acteurs très importants de la mise au point du langage Java (il fut également le créateur de BSD, la version de Berkeley de l'OS Unix). Il y a quelques années de cela, à l'orée du deuxième millénaire, il joua les Cassandre en rédigeant un article extrêmement pessimiste dans le célèbre magazine *Wired Magazine*, militant en faveur d'un « ralentissement » de la recherche scientifique. Son article, intitulé joyeusement « Pourquoi l'avenir n'a pas besoin de nous », se préoccupe des risques à long terme induits par le développement à tout crin actuel des nanotechnologies, de la génétique et de la robotique. En substance, il craint que de nouveaux artefacts issus de ces développements ne nous dépassent et n'échappent complètement à notre contrôle. C'est pas la « Joy ».

Aujourd'hui le langage Java en est à sa sixième version (Java 6), La cinquième version fut surtout remarquée pour l'introduction de modifications de base importantes comme la généricité, les énumérations ou encore de nouveaux types de boucle for que nous traiterons au chapitre 21.

Nous ne pourrions terminer cet encart sans vous indiquer une petite liste, loin d'être exhaustive bien sûr, et risquant l'obsolescence à la sortie du livre (si le choix se présente, optez toujours pour la dernière version de ces mêmes ouvrages), de nos références bibliographiques en programmation Java :

– *Cahier du programmeur Java 1.4 et 5.0*, Puybaret, Eyrolles 2004.
– *Au cœur de Java*, : vol. 1 et 2, HORSTMAN et CORNELL, CampusPress.
– *Comment programmer en Java*, DEITTEL et DEITTEL, Goulet, et autres ouvrages Java cosignés par ces deux auteurs.
– *Beginning Java 3*, Ivor HORTON, Wrox.
– *Thinking in Java 1 & 2*, Bruce ECKEL, www.mindview.net/Books/TIJ, Prentice Hall.
– *Java for Practitioners*, John HUNT, Springer Verlag.

La compilation Java : effet domino

L'interaction entre classes, ainsi que la modularisation recommandée de ces classes en fichiers, permettent d'obtenir l'impression d'écran présentée à la figure 5-1, qui correspond à la situation décrite ci-après.

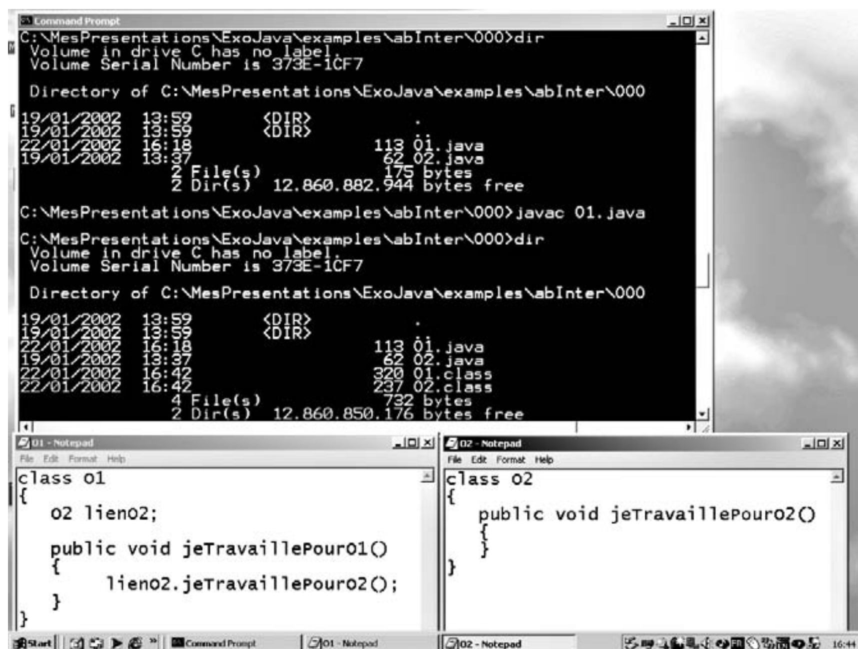


Figure 5-1

Deux fichiers Java contenant chacun une classe, et les liens dynamiques qui s'établissent lors de la compilation.

Deux fichiers Java séparés, `O1.java` et `O2.java`, contiennent, respectivement, l'un le code de la classe `O1`, l'autre le code de la classe `O2`. Dans la fenêtre DOS, vous les voyez apparaître tous deux dans leur répertoire. Ce sont deux fichiers source Java qui portent l'extension « java ». Nous compilons le premier à l'aide de l'instruction de compilation *javac :*

```
javac O1.java
```

Automatiquement, deux nouveaux fichiers apparaissent. Non seulement, comme prévu, l'exécutable issu du premier fichier : `O1.class`, mais également, de manière plus surprenante, la version exécutable du deuxième fichier : `O2.class`, alors que nous n'avons jamais demandé sa compilation. La raison en est, bien sûr, que le compilateur s'est aperçu que la classe `O1` nécessite pour sa réalisation la classe `O2`, et a, de ce fait, pris l'initiative heureuse de compiler aussi la classe `O2`. Dès que le compilateur découvre une dépendance entre les classes, il se charge de toutes les compilations nécessaires.

Cela tient de la magie, nous direz-vous ? Comment savait-il où trouver le code de la classe `O2 ?` Il manque un détail clé à l'explication. Nous avons nommé, comme il est classique de le faire en Java, le fichier du même nom que la classe. La classe `O2` se trouve dans le fichier `O2.java`, et le tour est joué, le compilateur sait maintenant comment trouver le code de la `classe O2`, afin de le relier à `O1`. Il en sera de même lors de l'exécution. En partant de la seule exécution du fichier contenant la méthode `main`, toutes les classes dépendantes entre elles seront reliées dynamiquement lors de cette exécution. La découpe et l'association une classe-un fichier découlent naturellement de ce mécanisme de liaison dynamique.

Vous constatez qu'il n'y a point besoin d'effectuer une liaison explicite entre les fichiers qui doivent se connaître mutuellement. Les classes ont besoin l'une de l'autre. En nommant les fichiers par le nom des classes qu'ils contiennent, automatiquement, les fichiers sauront comment se trouver et se lier, tant pendant la phase de compilation que pendant la phase d'exécution. Ce mécanisme de découverte automatique du code de la classe O2, en reprenant le nom de la classe pour le fichier, est propre à Java, et disparaît dans les autres langages de programmation OO, comme C++, C#, Python et PHP 5. Cela n'en reste pas moins une manière de procéder aussi élégante qu'efficace.

En Java, la classe, dans sa version exécutable .class, est toujours isolée dans un fichier, le nom du fichier devenant automatiquement le nom de la classe compilée qu'il contient. Même si, au départ, plusieurs classes sont codées dans un seul fichier source, la compilation créera autant de fichiers qu'il y a de classes distinctes. Cette modularisation forcée, mais parfaitement adéquate, disparaît des autres langages, pour des raisons de compatibilité avec les technologies les ayant précédés, souci non partagé par les ingénieurs de Sun, lesquels avaient décidé à l'époque de repartir de zéro.

Faire de chaque classe un fichier source séparé devient, de fait, une pratique tendant à se répandre dans tous les langages OO, que ces langages l'encouragent ou non par leur syntaxe et leur fonctionnement propres.

Une classe, un fichier

Dans sa pratique, et bien plus que les trois autres langages, Java force la séparation des classes en fichiers distincts. Si vous ne le faites pas lors de l'écriture des sources, il le fera pour vous, comme résultat de la compilation de ces sources. En conséquence de quoi, autant le précéder, par une écriture des classes séparée en fichiers. Cette bonne pratique tend à se généraliser à tous les développements OO, quel que soit le langage de programmation utilisé.

En C#, en Python, PHP 5 et en C++

En C#, il est nécessaire, pour que la classe `O1` (installée dans le fichier `O1.cs`) puisse se rattacher à la classe `O2` (installée dans le fichier `O2.cs`), de faire d'abord de la classe `O2` une librairie « dll » (*dynamic link library*, extension qui ne surprendra en rien les habitués de Windows), et ce au moyen de l'instruction suivante :

```
csc /t:library /out:O2.dll O2.cs
```

Ensuite, il faut compiler le fichier `O1.cs`, en faisant le lien avec le fichier dll généré précédemment :

```
csc /r:O2.dll O1.cs
```

Fichiers dll

Les fichiers portant l'extension .dll sont des fichiers caractéristiques des plates-formes Windows et qui permettent de relier dynamiquement plusieurs fichiers exécutables. C'est la raison pour laquelle, afin de rendre les nouveaux exécutables C# compatibles avec la plate-forme Windows, il faut en faire des fichiers .dll. Dans la plate-forme de développement .Net, ces fichiers .dll permettent de faire le lien entre n'importe quelles classes développées dans n'importe lequel des langages de programmation supportés par .Net (et ils sont nombreux puisqu'on en dénombre vingt-deux).

La situation en Python est telle qu'illustrée dans les deux fichiers qui suivent. Il est nécessaire d'indiquer explicitement dans les premières lignes du fichier `O1` où trouver les classes dont il aura besoin.

```
Fichier O1.py

from O2 import *      # rapelle les classes du fichier O2
class O1:
    __lienO2=O2()
    def jeTravaillePourO1(self):
        __lienO2.jeTravaillePourO2(5)


Fichier O2.py

class O2:
    def jeTravaillePourO2(self,x):
        print x
```

En C++ aussi, il est nécessaire d'inclure dans le fichier `O1.cpp`, l'instruction d'inclusion du fichier `O2.cpp`, comme dans le code qui suit :

```
#include "O2.cpp" /* inclusion de la classe dont la nouvelle classe dépend */
class O1 {
private:
  O2* lienO2;
public:
  void jeTravaillePourO1() {
    lienO2->jeTravaillePourO2();
  }
};
```

Nous retrouvons le même type d'inclusion dans la version PHP 5 du même code.

```
<html>
<head>
<title> Association de classes </title>
</head>
<body>
<h1> Association de classes </h1>
<br>

<?php
   include ("O2.php");
   class O1 {
            private $lienO2;
            public function __construct() {
                     $this->lienO2 = new O2();
            }
            public function jeTravaillePourO1() {
                     $this->lienO2->jeTravaillePourO2();
            }
   }
   $unO1 = new O1();
   $unO1->jeTravaillePourO1();

?>

</body>
```

```
</html>

Avec dans le même répertoire Web, le fichier O2.php.

<?php
    class O2 {
                public function jeTravaillePourO2() {
                            print ("je travaille pour O2 <br> \n");
                }
    }
?>
```

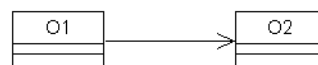
Dans tout langage, la liaison dynamique entre les classes exige des « liants syntaxiques » supplémentaires et ne se réalise plus implicitement, à l'instar de Java, comme simple résultat de la dénomination des classes et des fichiers.

De l'association unidirectionnelle à l'association bidirectionnelle

Une question assez légitime peut être posée, quand on s'aperçoit que la liaison dynamique à la compilation se fait, soit en ordonnant les compilations, comme en C#, à savoir d'abord en compilant le premier fichier dont dépend le second, ensuite le second, soit, comme en Python, PHP 5 et C++, par l'inclusion du premier dans le second. Que se passe-t-il quand les deux classes dépendent mutuellement l'une de l'autre ?

Dans les petits diagrammes de classe UML suivants, vous pouvez voir la différence entre une association de type directionnelle et une association bidirectionnelle. Ces dernières associations sont très fréquentes dans la conception OO. Prenez, par exemple, l'association entre un employé et un employeur, un joueur de foot et son capitaine, la proie et le prédateur, un ordinateur et son imprimante, etc. L'association entre deux classes est bidirectionnelle quand des messages peuvent être envoyés dans les deux sens.

Figure 5-2
Différence entre une association de deux classes de type directionnelle et de type bidirectionnelle.



association directionnelle

association bidirectionnelle

En Java, cette situation ne change absolument rien à la pratique décrite plus haut. La compilation de l'une des deux classes entraînera automatiquement, dans sa suite, la compilation de l'autre.

En C#, comme l'ordre de compilation est déterminé par les liens de dépendances entre les classes, la situation est plus délicate, et la solution la plus immédiate, parmi d'autres, consistera à compiler les fichiers, tous ensemble, et non plus de manière séparée.

En C++, c'est l'écriture du code qu'il faudra soigner afin de pallier cette situation. Il faudra séparer la déclaration des classes de la description de leur corps. Cette description devra être différée par rapport à la seule déclaration des classes, comme le code ci-dessous en est l'illustration.

Fichier O2.cpp

```
#include "iostream.h"
class O1;  /* juste la déclaration de la classe et rien d'autre */
class O2 {
private:
  O1* lienO1;
public:
  void jeTravaillePourO2();  /* la méthode sera définie plus tard */
};
```

Fichier O1.cpp

```
#include "iostream.h"
#include "O2.cpp"
class O2;  /* juste la déclaration de la classe et rien d'autre */
class O1 {
private:
  O2* lienO2;
public:
    void jeTravaillePourO1();  /* la méthode sera définie plus tard */
};
/* puis enfin, la description du contenu des méthodes */
void O1::jeTravaillePourO1() {
  lienO2->jeTravaillePourO2();
}
void O2::jeTravaillePourO2() {
  lienO1->jeTravaillePourO1();
}
int main() {
  cout << "ca marche" << endl;
  return 0;
}
```

Enfin, comme Python et PHP 5 sont tout deux des langages de script, c'est-à-dire s'exécutant directement, sans phase préalable de compilation, au fur et à mesure que les instructions sont rencontrées, on se rend compte de l'impossilité produite par cette référence (ici importation) circulaire. Lorsque l'exécution de la première classe s'interrompra pour importer la deuxième, et que cette deuxième ne pourra pas non plus s'exécuter faute de la première, on se trouvera coincé dans une situation sans issue. La solution la plus simple est de contourner le problème, de ne réaliser l'inclusion que dans la première classe et prévoir une méthode dans la deuxième pour relier celle-ci à la prémière. Dans l'exemple PHP 5 ci-après, c'est la solution qui est proposée. Notez dans cet exemple que, malgré l'absence de typage dans PHP 5, il est cependant possible, lorsque les arguments de méthode concernent des classes, de le spécifier dans la déclaration. Lors de l'appel, un mauvais passage d'arguments donnera une erreur fatale.

Premier fichier PHP contenant la classe `O1` :

```
<html>
<head>
<title> Association de classes </title>
</head>
<body>
<h1> Association de classes </h1>
<br>

<?php
   include ("O2-2.php");
   class O1 {
          private $lienO2;
          public function __construct(O2 $unO2) { // notez le typage de l'argument du constructeur
                   $this->lienO2 = $unO2;
          }
          public function jeTravaillePourO1() {
                   print("je travaille pour O1 <br> \n");
                   $this->lienO2->jeTravaillePourO2(); // envoi de message vers la classe O2
          }
          public function jeTravailleAussiPourO1() {
                   print("je travaille aussi pour O1 <br> \n");
          }
   }

   $unO2 = new O2();
   $unO1 = new O1($unO2);
   $unO2->setO1($unO1);
   $unO1->jeTravaillePourO1();
   $unO2->jeTravaillePourO2();

?>

</body>
</html>
```

Deuxième fichier PHP `O2-2.php` contenant la classe `O2` :

```
<?php
    class O2 {
          private $lienO1;
          public function jeTravaillePourO2() {
                   print ("je travaille pour O2 <br> \n");
                   $this->lienO1->jetravailleAussiPourO1(); // envoi de message vers la classe O1
          }

          public function setO1(O1 $unO1) { // la méthode réalise l'association avec la première classe
                   $this->lienO1 = $unO1;
          }
    }
?>
```

Auto-association

Une dernière chose : les classes peuvent bien évidemment s’adresser à elles-mêmes, en devenant les destinataires de leur propre message, comme montré dans le diagramme ci-après.

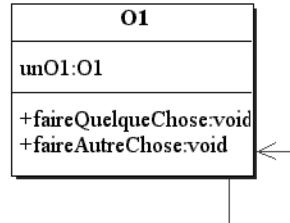


Figure 5-3
La classe en interaction avec elle-même.

Lors de l’exécution d’une de ses méthodes, un objet peut demander à une autre de ses méthodes de s’exécuter. Il s’agit du mécanisme classique d’appel imbriqué de méthodes, comme indiqué dans le petit code suivant, dans lequel le corps de la méthode `faireQuelqueChose()` intègre un appel à exécution de la méthode `faireAutreChose()`.

```
faireQuelqueChose(int a) {
    …
    faireAutreChose() ;
}
```

Nous verrons dans les chapitres 7 et 8 consacrés à la pratique de l’encapsulation que, si la méthode `faireAutreChose` est déclarée comme `private`, elle ne pourra jamais être appelée autrement qu’à l’intérieur d’une autre méthode de la même classe.

> **Appel imbriqué de méthodes**
>
> On imbrique des appels de méthodes l’un dans l’autre quand l’approche procédurale se rappelle à notre bon souvenir. Force est de constater que l’OO ne se départ pas vraiment du procédural. L’intérieur de toutes les méthodes est, de fait, programmé en mode procédural comme une succession d’instructions classiques, assignation, boucle, condition… L’OO vous incite principalement à penser différemment la manière de répartir le travail entre les méthodes et la façon dont les méthodes s’associeront aux données qu’elles manipulent, mais ces manipulations restent entièrement de type procédural. Ces imbrications entre macrofonctions sont la base de la programmation procédurale, ici nous les retrouvons à une échelle réduite, car les fonctions auront préalablement été redistribuées entre les classes.

Plus généralement, tout objet d’une classe donnée peut contenir dans le corps d’une de ses méthodes un appel de méthode à exécuter sur un autre objet, mais toujours de la même classe, comme dans le code qui suit :

```
class O1{
  O1 unAutreO1 ;
  faireQuelqueChose(){
    unAutreO1.faireAutreChose();
  }
}
```

Un joueur de football peut faire une passe à un autre joueur. Le prédateur peut partir à la recherche d’un autre prédateur.

Les diagrammes de séquence UML qui suivent montrent la différence entre les deux cas, différence qui se marque dans le destinataire du message, dans le premier cas, l'objet lui-même, dans le second cas, un autre objet, mais de la même classe.

Figure 5-4
Envoi de message à l'objet.

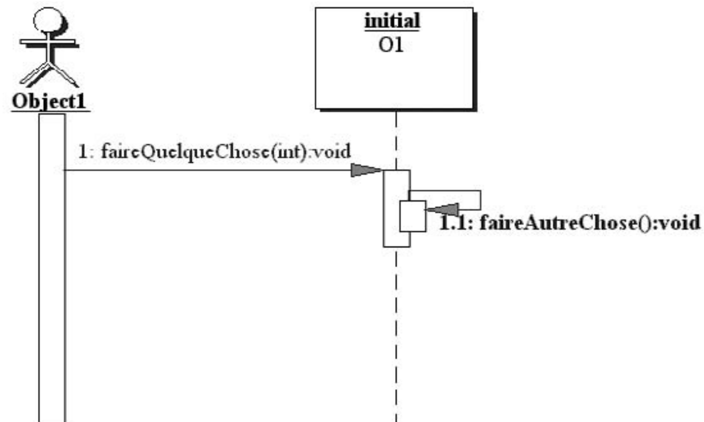
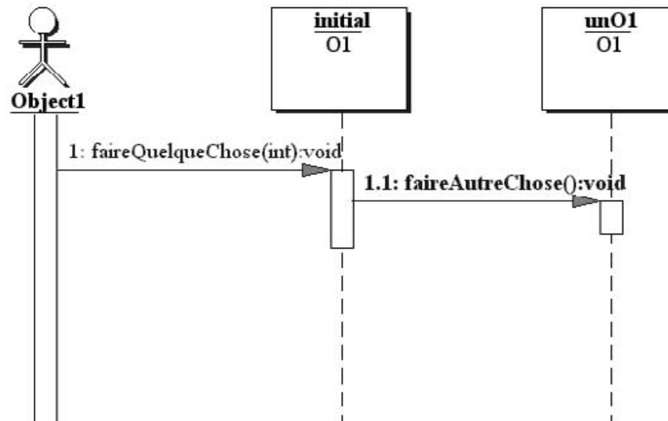


Figure 5-5
Envoi de message à un autre objet, de la même classe.



Alors qu'il s'agira, contrairement au cas précédent, d'un transfert de message entre deux objets différents, du point de vue des classes, il s'agira d'une interaction entre une classe et elle-même. Cela se produira dans notre petit exemple de l'écosystème, si les prédateurs ou les proies veulent communiquer, entre prédateurs et entre proies.

Package et namespace

Comme nous l'avons vu dans le chapitre 2, au même titre que vous organisez vos fichiers dans une structure hiérarchisée de répertoires, vous organiserez vos classes dans une structure isomorphe de paquetage (*package* en Java et Python, *namespace* en C++, C# et PHP 5). Il s'agit là, uniquement, d'un mécanisme de nommage

de classes, comme les répertoires le sont pour les fichiers, et qui vous permet, tout à la fois, de regrouper vos classes partageant un même domaine sémantique, et de donner un nom identique à deux classes placées dans des packages différents.

En Java, le système de nommage des classes doit s'accompagner de l'emplacement des fichiers dans les répertoires correspondants. Supposez par exemple que la classe `O2` nécessaire à l'exécution de la classe `O1` soit dans un paquetage `O2`, comme indiqué dans le code qui suit. Tant le code source de la classe `O2` que son exécutable devront se trouver dans le répertoire `O2`. La classe `O1`, elle, se trouvera juste un niveau au-dessus.

Fichier O2.java

```
/* Ce fichier ainsi que le fichier .class devront être placés dans
         le répertoire O2 */
package O2;
public class O2 {
   public void jeTravaillePourO2() {
     System.out.println("Je travaille pour O2");
  }
}
```

Fichier O1.java

```
/* Ce fichier ainsi que le fichier .class devront être placés dans
         le répertoire juste au-dessus d'O2 */
import O2.*; /* Pour rappeler les classes du répertoire O2 */
public class O1 {
   public void jeTravaillePourO1(){}
   public static void main(String[] args) {
     O2 unO2 = new O2(); /* Il ne s'agit là que d'un système de nommage des classes
         En lieu et place de l'import, on pourrait renommer la classe O2 par O2.O2.*/
     unO2.jeTravaillePourO2();
   }
}
```

En C#, en revanche, tout comme en C++ et PHP 5, le namespace n'est qu'un système de nommage hiérarchisé de classes, sans nécessaire correspondance avec l'emplacement des fichiers dans les répertoires. Nommage des fichiers et nommage des classes deviennent donc indépendants. Ainsi, les deux fichiers C# qui suivent peuvent parfaitement se retrouver dans un même répertoire, tant dans leur version source que compilée, malgré la présence de `namespace` dans l'un et de `using` dans l'autre.

Fichier O2.cs

```
/* Le namespace et la classe doivent différer dans leur nom */
namespace O22
{
 public class O2 {
   public void jeTravaillePourO2() {
     System.Console.WriteLine("Je travaille pour O2");
  }
}
}
```

Fichier O1.cs

```
using O22;
public class O1 {
   public void jeTravaillePourO1(){}
   public static void Main() {
     O2 unO2 = new O2();/* Il ne s'agit là que d'un système de nommage des classes
            En lieu et place du using, on pourrait renommer la classe O2 par O22.O2
     unO2.jeTravaillePourO2();
   }
}
```

En débutant un programme Java par l'instruction `import ….` et en C# par `using …`, vous signalez que, tant durant la compilation que l'exécution du programme, les classes qui y sont référées, si elles sont absentes du répertoire courant, sont à rechercher dans les paquetages mentionnés dans ces deux instructions.

> **`import` en Java et `using` en C#**
>
> Vos classes étant regroupées en paquetages imbriqués, il est indispensable, lors de leur utilisation, soit de spécifier leur nom complet : « Paquetage.classe » (d'abord le nom du paquetage puis le nom de la classe), soit d'indiquer, au début du code, le paquetage qui doit être utilisé afin de retrouver les classes exploitées dans le fichier. Cela se fait par l'addition, au début des codes, de l'instruction `import` en Java et `using` en C#.

Finalement en Python, un mécanisme de paquetage est également possible, comme en Java, en totale correspondance avec les répertoires. Supposons le fichier `O2.py` contenant la classe `O2` et placée dans le répertoire `O2`. En matière de classe, il s'agira donc du paquetage `O2`. Pour que la classe `O1` puisse disposer des classes contenues dans le paquetage, il suffit d'inclure la commande `from O2.O2 import *` en début de fichier. Il faudra également, truc et ficelle, inclure un fichier vide et dénommé `__init__.py` dans le répertoire `O2` en question.

Fichier O2.py

```
# placé dans le répertoire O2
   class O2:
    def jeTravaillePourO2(self,x):
      print x
   }
```

Fichier O1.py

```
from O2.O2 import *
class O1:
    __lienO2=O2()
    def jeTravaillePourO1(self):
        __lienO2.jeTravaillePourO2(5)
print "ca marche"
```

Finalement, dans tous les cas, la représentation UML de cette situation à l'aide d'un diagramme UML dit de « package » (la classe `O1` associée à la classe `O2` se trouvant dans le paquetage `O22`) est illustrée par la figure 5-6.

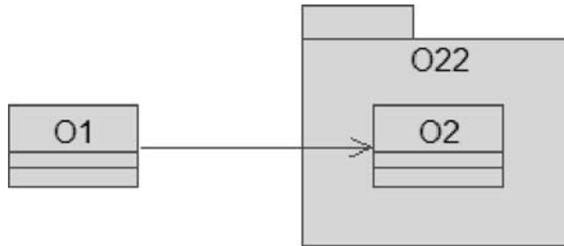


Figure 5-6
La classe 01 envoie un message à la classe 02 placée dans un paquetage 022.

Exercices

Exercice 5.1

Revenez à l'analyse orientée objet du premier exercice du chapitre 1, consistant en une recherche des classes décrivant votre activité favorite. Approfondissez la nature des relations existant entre les classes et prenez soin de différencier des relations d'auto-association, des associations directionnelles ou bidirectionnelles.

Exercice 5.2

Toujours dans la description OO que vous faites de cette activité, réfléchissez à une nouvelle organisation des classes en assemblage. Quelles classes installeriez-vous dans un même assemblage, et quelle structure imbriquée d'assemblage pourriez-vous réaliser ?

Exercice 5.3

Écrivez le code d'une classe s'envoyant un message à elle-même, d'abord lorsque ce message n'implique qu'un seul objet, ensuite lorsque ce message en implique deux.

Exercice 5.4

Écrivez le code d'une classe A qui, lors de l'exécution de sa méthode `jeTravaillePourA()`, envoie le message `jeTravaillePourB()` à une classe B. Séparez les deux classes dans deux fichiers distincts et, quel que soit le langage que vous utilisez, réalisez l'étape de compilation.

Exercice 5.5

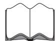
Sachant que la classe A est installée dans l'assemblage `as1`, lui-même installé dans l'assemblage `as`, quel est le nom complet à donner à votre classe ?

6

Méthodes ou messages ?

Ce chapitre aborde de manière plus technique les mécanismes d'envoi de message. Les passages d'argument par valeur ou par référent, qu'il s'agisse de variables de type prédéfini ou de variables objet, sont discutés dans le détail et différenciés dans les cinq langages. La différence entre un message et une méthode est précisée. La notion d'interface et le fait que les messages puissent circuler à travers Internet sont, à ce stade, simplement évoqués.

Sommaire : Passage d'arguments dans les méthodes — Passage par valeur et par référent — Passage d'objets — Méthodes et messages — Introduction aux interfaces et aux objets distribués



Candidus — J'aimerais maintenant savoir ce qui se cache derrière les boutons de commande de nos objets. Je sais bien qu'ils actionnent leurs différentes fonctions mais tu appelles ça des messages. Pourquoi ce nouveau terme, d'ailleurs ?

Doctus — Parce qu'il s'agit bien de messages. Même dans le cas des langages procéduraux qui ne connaissent que le seul domaine global d'un programme, tu peux voir les appels de fonctions comme des messages envoyés à un objet unique que constitue le programme lui-même.

Cand. — Alors nos objets prennent des initiatives ? Ce sont des objets communicants !

Doc. — C'est exact, créer un objet consiste en tout premier lieu à définir son vocabulaire, ce qu'il peut « comprendre », à savoir l'ensemble des messages qu'il peut traiter. On appelle ça son interface. Bébé pourra jouer avec certains boutons et les objets eux-mêmes joueront les uns avec les autres de la même façon.

Cand. — Si j'ai bien observé, certains boutons messages doivent être actionnés à l'aide d'accessoires. Il me semble y reconnaître les paramètres de nos fonctions procédurales. Que se passe-t-il exactement quand un message est envoyé à un objet ?

Doc. — Tout d'abord, un message écrit par une main humaine sur du papier contient des informations qui ne sont que la copie d'une partie de ce qui se trouve dans le cerveau de son auteur.

Cand. — C'est malin ! J'aurais pu trouver ça tout seul…

Doc. — …je continue ! Un message peut aussi contenir le moyen d'accéder à d'autres informations que celles qu'il contient…

Cand. — Une clé par exemple ?

Doc. — Exactement. La dénomination adoptée pour cette clé est référent, qui peut n'être qu'une adresse.

Cand. — Je vois où tu veux en venir ! L'objet appelant a donc le choix entre donner une copie de ses informations et donner le moyen d'y accéder. Est-ce bien ça ?

Doc. — C'est bien ça, la différence étant que l'accès à une source d'informations permet d'en changer la valeur, tandis qu'une simple copie ne le permet pas.

Cand. — Et quelle est la distinction entre message et méthode ?

Doc. — On appelle méthode ce qu'un objet exécute lorsqu'il reçoit le message associé. Les envois de message, eux, correspondent aux appels de fonction.

Cand. — Je pense que je vais retrouver mes marques en passant à l'OO. Mes fonctions, leurs arguments et leur valeur de retour éventuelle... Il ne s'agit en fait que d'un pas supplémentaire vers la distribution des tâches.

Doc. — Attention ! Le choix de passer une copie ou une référence n'est pas disponible de manière identique dans tous les langages. Un argument de message peut être lui-même constitué par un objet. Je te suggère de réfléchir à ce que cela implique quant aux possibilités de concevoir des systèmes beaucoup plus ouverts à la créativité que ce que nous permettent les langages procéduraux.

Passage d'arguments prédéfinis dans les messages

Pour envoyer un bon message, procédez avec méthode. En effet, les objets se parlent par envois de message, lorsqu'un objet passe la main à un autre, afin qu'une méthode s'exécute sur ce dernier. Lors de son exécution, comme en programmation classique, la méthode peut recevoir des arguments. Ces arguments seront utilisés dans son corps d'instruction. Ces arguments, tout comme lorsqu'on déclare une fonction mathématique f(x), permettent d'affiner ou de calibrer le comportement de la méthode, en fonction de la valeur de l'argument passé. Considérons à nouveau la déclaration de la `méthode jeTravaillePourO2(int x)` de la classe `O2`, mais qui, cette fois, prévoit de recevoir un argument de type entier : « x ». Cette méthode peut être activée par un message, comme dans le petit exemple suivant :

```
class O1 {
  O2 lienO2 ;
  void jeTravaillePourO1() {
     lienO2.jeTravaillePourO2(5) ;
  }
}
```

Rien de particulier n'est à signaler. Ajoutons maintenant, que la méthode `jeTravaillePourO2(int x)` modifie l'argument qu'elle reçoit, comme dans le petit code Java ci-après. Tâchez, sans regarder le résultat, de prévoir ce qui sera produit à l'écran.

En Java

```
class O2 {
  void jeTravaillePourO2(int x) {
    x++; /* incrément de l'argument */
    System.out.println("la valeur de la variable x est: " + x);
  }
}
```

```
public class O1 {
  O2 lienO2;
  void jeTravaillePourO1() {
    int b = 6;
    lienO2 = new O2();
    lienO2.jeTravaillePourO2(b);
    System.out.println("la valeur de la variable b est: " + b);
  }
  public static void main(String[] args) {
    O1 unO1 = new O1();
    unO1.jeTravaillePourO1();
  }
}
```

Résultat

```
la valeur de la variable x est : 7

la valeur de la variable b est : 6
```

Qu'advient-il de la variable locale `b`, créée dans la méthode `jeTravaillePourO1()`, et passée comme argument du message `jeTravaillePourO2(b)`? Sa nouvelle valeur sera-t-elle 7 ? Non, car en général, un passage d'argument s'effectue de manière préférentielle « par valeur ».

On entend par là la création d'une variable temporaire `x`, recopiée de l'originale, qui recevra, le temps de l'exécution de la méthode, la même valeur que la valeur transmise : 6, et disparaîtra à la fin de cette exécution. La variable de départ `b` est laissée complètement inchangée, seule la copie est affectée. L'exécution de la méthode s'accompagne, en fait, d'une petite mémoire pile (dernier entré premier sorti), dont le temps de vie est celui de cette exécution, pas une seconde de plus. Alors que c'est l'unique type de passage permis par Java, d'autres langages ont enrichi leur offre. Lisez avec attention le code C# suivant et tentez, là encore, de prédire son résultat.

En C#

```
using System;
class O2 {
  public void jeTravaillePourO2(int x) {
    x++;
    Console.WriteLine("la valeur de la variable x est: " + x);
  }
  public void jeTravaillePourO2(ref int x) /* observez bien l'addition du mot-clé ref */ {
    x++;
    Console.WriteLine("la valeur de la variable x est: " + x);
  }
}
public class O1 {
  O2 lienO2;

  void jeTravaillePourO1() {
    int b = 6;
    lienO2 = new O2();
    lienO2.jeTravaillePourO2(b);
    Console.WriteLine("la valeur de la variable b est: " + b);
    lienO2.jeTravaillePourO2(ref b); /* observez bien l'addition du mot-clé ref */
    Console.WriteLine("la valeur de la variable b est: " + b);
```

```
    }
    public static void Main() {
      O1 unO1 = new O1();
      unO1.jeTravaillePourO1();
    }
  }
```

Résultat

```
la valeur de la variable x est : 7
la valeur de la variable b est : 6
la valeur de la variable x est : 7
la valeur de la variable b est : 7
```

Nous avons, dans le code C#, déclaré deux fois la méthode `jeTravaillePourO2(int x)`, la première fois, comme en Java, la seconde fois en spécifiant que nous voulions effectuer le passage d'arguments par référent. Nous utilisons ici le mécanisme de surcharge, discuté dans le chapitre 2, qui permet l'utilisation de deux méthodes différentes, bien que nommées de la même manière. Dans le second cas, ce n'est plus la valeur de la variable que nous passons, mais bien une copie de son référent qui, tout comme le référent d'un objet, contient l'adresse de la variable. En modifiant cette variable, on modifiera cette fois la valeur contenue à cette adresse, en conséquence, la variable de départ elle-même, et non plus une copie de celle-ci.

En C++

C++ vous permet, à l'aide d'une écriture un peu plus déroutante, d'y parvenir également. Afin de comprendre le code présenté ci-après, il faut savoir que, lorsqu'une variable est déclarée comme pointeur : `int *x`, on autorise l'accès direct à son adresse, adresse contenue dans `x`. On peut, de surcroît, modifier cette adresse, et faire pointer le pointeur vers un autre espace mémoire. Il suffit d'écrire par exemple `x++`. C'est un jeu évidemment très dangereux dont la pratique entame la réputation du C++ en matière de sécurité. La valeur pointée par le pointeur, quant à elle, est obtenue en écrivant `*x`.

De même, il est toujours possible d'obtenir l'adresse d'une quelconque variable `y`, en écrivant, simplement, `&y`. Mais il ne sera jamais possible d'écrire une instruction comme `&y++`, qui permettrait de modifier cette adresse. Ce qu'on appelle un référent en C++, pour le différencier d'un pointeur, et indiqué par la présence de `&`, référera toujours une seule et même adresse.

```
#include "iostream.h"
class O2 {
public:
  void jeTravaillePourO2(int x){
    x++;
    cout << "la valeur de la variable x est: " << x << endl;
  }
  /*  void jeTravaillePourO2(int &x)  elle ne peut fonctionner en même temps que la première version
      ➥ car son appel serait alors ambigu {
    x++;
    cout << "la valeur de la variable x est: " << x << endl;
  }*/
  void jeTravaillePourO2(int *x) /* on peut, par cette nouvelle signature, surcharger la première
  ➥ version */ {
```

```
    ++*x;  /* Si vous écrivez *x++, vous serez surpris du
        résultat, car l'incrément se fera sur l'adresse et
        non plus la valeur */
    cout << "la valeur de la variable x est: " << *x << endl;
  }
};
class O1 {
  O2 *lienO2;
public:
  void jeTravaillePourO1() {
    int b = 6;
    lienO2 = new O2();
    lienO2->jeTravaillePourO2(b); /* appelle de manière semblable la première version ou la deuxième
    ➥ version, d'où l'impossibilité d'une déclaration commune */
    cout << "la valeur de la variable b est: " << b << endl;
    lienO2->jeTravaillePourO2(&b); /* n'appelle que la troisième version */
    cout << "la valeur de la variable b est: " << b << endl;
  }
};
int main() {
    O1 unO1;
    unO1.jeTravaillePourO1();
    return 0;
}
```

Résultat

```
la valeur de la variable x est : 7
la valeur de la variable b est : 6
la valeur de la variable x est : 7
la valeur de la variable b est : 7
```

Dans la classe `O2`, la première version de la méthode `jeTravaillePourO2()` fonctionne comme en Java, et le passage d'argument se fait par valeur. Deux options sont alors proposées pour effectuer le passage d'argument par référent. La première, celle qui est usuellement recommandée, effectue un réel passage par référent (en utilisant la notation `&`), car il s'agit bien de l'adresse qui est passée. Mais, comme vous pouvez le voir dans le code, vous ne pouvez utiliser cette seconde version en même temps que la première (celle pour qui le passage d'argument se fait par valeur) car, lors de l'appel, il est impossible de différencier laquelle des deux est à exécuter.

La seconde version (en fait la troisième définition de la méthode) utilise, elle, un pointeur pour recevoir l'adresse de la variable. Vous la rencontrerez moins souvent. Ces écritures sont assez laborieuses et sont à l'origine de nombreux maux de têtes et mal-être existentiels dans notre communauté informatique. Les psychanalystes et autres psychiatres, depuis des années, se battaient pour en interdire l'usage. Java et Python les ont entendus. Ils l'ont fait.

En Python

En Python, le code qui suit vous permet de comprendre aisément que le seul passage d'argument autorisé est par valeur. On constatera encore l'absence de typage, puisqu'il n'est pas nécessaire de spécifier le type des arguments lors de la définition des méthodes. Au moment de l'appel de la méthode, le type dépendra automatiquement de la valeur transmise. Dernière observation en passant : remarquez combien ce code est plus court

et plus simple à écrire que son équivalent en Java, surtout du côté du `main`, 12 lignes pour Python contre 19 en Java. Qui dit mieux ? La brièveté et la simplicité d'écriture de Python (comme pour pas mal de langages de script, à l'instar de PERL ou de PHP, par exemple) sont des arguments que l'on avance souvent en sa faveur. En fait, ces langages reprennent à leur compte le genre d'arguments que Java a dû avancer pour s'imposer devant C++. Ils ont bien retenu la leçon.

En Python

```
class O2:
      def jeTravaillePourO2(self,x):
               x+=1
               print "la valeur de la variable x est: %s" % (x)
class O1:
      def jeTravaillePourO1(self):
               b=6
               lienO2=O2()
               lienO2.jeTravaillePourO2(b)
               print "la valeur de la variable b est: %s" % (b)
unO1=O1()
unO1.jeTravaillePourO1()
```

Résultat

```
la valeur de la variable x est : 7
la valeur de la variable b est : 6
```

En PHP 5

En PHP 5, comme en C++, les deux passages, par valeur et par référent, sont acceptés selon que l'on ajoute le & oui ou non, lors de la déclaration de l'argument. Le code ci-après est la version PHP 5 des passages de code précédents et seule l'addition du &, comme indiqué à même le code, modifie le résultat.

```
<html>
<head>
<title> Passage d'arguments </title>
</head>
<body>
<h1> Passage d'arguments </h1>
<br>

<?php
   class O2 {
       public function jeTravaillePourO2(&$x){ // Selon que l'on ajoute ou non le &, le résultat
       ➥sera 6 ou 7.
             $x+=1;
             print ("la valeur de la variable x est $x <br> \n");
       }
   }
```

```
    class O1 {
        public function jeTravaillePourO1(){
              $b=6;
              $lienO2 = new O2();
              $lienO2->jeTravaillePourO2($b);
              print ("la valeur de la variable b est $b <br> \n");
        }
    }

    $unO1 = new O1();
    $unO1->jeTravaillePourO1();

?>

</body>
</html>
```

Passage par valeur ou par référent

En ce qui concerne le passage d'arguments de type prédéfini, le passage par valeur aura pour effet de passer une copie de la variable et laissera inchangée la variable de départ, alors que le passage par référent passera la variable originale, sur laquelle la méthode pourra effectuer ses manipulations.

C++ et Bjarne Stroustrup

Première difficulté : écrire son nom sans se tromper, pour ne pas parler de la prononciation. Il en va souvent ainsi des Danois, même s'ils vivent depuis longtemps aux États-Unis, comme c'est le cas du créateur du C++. Stroustrup est actuellement professeur à la Texas A&M University et maintient de nombreuses collaborations avec son ancien lieu de travail : le laboratoire d'AT&T Bell (dans le New Jersey). C++ a été pendant longtemps le langage de programmation OO le plus populaire et le plus pratiqué (de nombreux logiciels Microsoft et bien d'autres tels que Netscape ont été programmés en C++, il est à la base de technologies COM et CORBA, décrites au chapitre 16, Stroustrup recence sur son site web des milliers de fameuses applications informatiques programmées dans ce langage), même s'il est en passe d'être détrôné aujourd'hui par Java et peut-être demain par C#. La syntaxe du C++ est le reflet de son histoire et de son évolution, l'union des langages C et Smalltalk ou Simula. Notre auteur ne voulait rien, ou si peu, abandonner du C, tout en greffant une couche OO sur ce même langage. Il souhaitait garder la compatibilité « descendante » avec le C. Or, d'une certaine manière, les objectifs du C et de la programmation OO sont, comme nous espérons vous en avoir convaincu, quelque peu contradictoires : C tend à coller au mieux à l'architecture et au fonctionnement des processeurs actuels, alors que l'OO tend à coller au mieux aux structures mentales et au fonctionnement cognitif des programmeurs. Dur de faire optimal et simple à la fois, car ce qu'on gagne d'un côté, on se trouve contraint et forcé de le perdre ailleurs. OO tire la couverture de la programmation vers le programmeur, C vers le processeur, situation quelque peu schyzophrénique, comme le deviennent de nombreux programmeurs dans ce langage.

La parade orale favorite de Stroustrup, face à la critique classique et sempiternelle adressée au C++ de langage hybride, est de retourner ce même argument en sa faveur, en n'en faisant, non plus un langage hybride, mais multi-paradigmatique, s'accommodant de plusieurs styles de programmation : procédural, numérique ou OO. Il met en avant les désavantages de la programmation objet quand le souci premier est la performance, économie mémoire et temps calcul, extrêmement critique pour le développement des applications dites embarquées. Il est indéniable que C++ est le langage le plus riche au travers des possibilités qu'il offre aux programmeurs. On voit clairement un petit noyau de Java ou de C# poindre dans C++, et certainement pas l'inverse.

Une façon succincte de différencier Java et C++ (attendons un peu pour placer C# dans cet exercice comparatif, car il se situe quelque part au milieu) serait de dire que la difficulté en Java ne réside pas tant dans la syntaxe de base du langage que dans la quantité énorme de librairies qu'il faut maîtriser pour des applications système, et que Java met à disposition dans l'environnement JDK. En revanche, la difficulté en C++ reste à ce jour principalement cantonnée dans la maîtrise de la syntaxe de base. Notez qu'une évolution actuelle du C++ est de privilégier davantage le développement des bibliothèques standards que l'évolution de la syntaxe du langage. Parmi ces nouvelles librairies, on retrouve, en commun avec Java, C#, et Python des utilitaires pour le multithreading, la persistance, la gestion automatique de la mémoire, etc. Pour l'instant, le programmeur C++ est, soit tenu de programmer l'utilitaire désiré, soit de se reposer entièrement sur les outils mis à sa disposition par le système d'exploitation (mais qui crée une forte dépendance avec la plate-forme sur laquelle tourne le programme) ou sur des développeurs occasionnels, soucieux d'enrichir les fonctionnalités du C++.

Enfin, en plus de la couche OO additionnelle, Stroustrup accorde une immense importance au mécanisme de généricité, absent de Java (mais intégré à partir de la version 1.5) et C# (mais intégré à partir de la version 2) et que nous abordons très brièvement dans le chapitre 21 lors de la programmation des listes liées, des graphes et des collections en général. Jusqu'alors, ces deux langages compensaient en partie l'absence de générécité par l'existence de la superclasse objet. La généricité permet de programmer à un très haut niveau d'abstraction, et de récupérer ces programmes pour de multiples objets, quel que soit leur type, sans se préoccuper de problème de surcharge de méthode ou de « casting » inapproprié. Le « casting » est en effet une pratique intéressante de la programma-tion car, alors que Stroustrup la considère comme « tordue » et à éviter (n'oublions pas la force du typage statique en C++), il est omniprésent dans des langages comme Java et C#, du fait du typage dynamique et de la pratique du polymorphisme. Son apect nuisible (il peut entraîner des erreurs à la phase d'exécution du code si le type dynamique de l'objet passé n'est pas celui prévu à la compilation) a incité les développeurs des langages Java et C# (et suivant en cela les critiques et les recommandations de Stroustrup) à en restreindre l'utilisation dans la nouvelle version du language. Vous l'aurez compris, Stroustrup privilégie la phase de compilaton et les assurances qui en découle pour la qualité et l'efficacité des codes. Il ne doit pas trop se retrouver dans l'engouement actuel pour les langages de script tels Python et PHP, qui s'en sont totalement émancipés.

Parmi nos manuels de programmation favoris en C++, indiquons (une liste très subjective), en commençant par les ouvrages du maître :

- *The C++ Programming Language*, Addison-Wesley. Plusieurs versions sont disponibles, mais vous aurez compris qu'une seule suffit, la plus récente. Sachez également que C++ est un standard *de facto* depuis 1998, et que, depuis lors, aucune nouvelle fonctionnalité significative n'a été rajoutée au langage.
- *Practical C++*, Rob McGregor, QUE.
- *C++ Primer Edition*, Lippman et Lajoie, Addison-Wesley.
- *More effective C++*, Scott Meyers, Addison-Wesley – un livre pour les pros (qui le conseillent plus), mais illustrant à souhait la richesse et la subtilité de la programmation en C++.
- *Mieux programmer en C++*, Herb Sutter (trad. Thomas Pétillon), Eyrolles (un excellent livre pour maîtriser des concepts avancés jusqu'aux moindres subtilités du C++).
- *Le langage C++*, Jacques Charbonnel, Masson.

Enfin, deux excellents ouvrages comparant C++ et Java :

- *Apprendre Java et C++ en parallèle*, Jean-Bernard Boichat, Eyrolles.
- *C++ for Java Programmers*, Timothy Budd, Addison-Wesley..

Nous nous en voudrions enfin de ne pas citer la nouvelle bouture C++ de Microsoft, le Visual C++ .Net, une espèce d'hybride fascinant (surtout pour les problèmes de gestion de la mémoire des objets) entre Java et C++, et certainement un des langages de programmation les plus puissants à ce jour. Microsoft doit en effet beaucoup à C++, au point qu'il aurait pu le renommer $++.

Passage d'argument objet dans les messages

Supposons maintenant que l'argument transmis à la méthode `jeTravaillePourO2(O3 lienO3)` soit un argument de type, non plus prédéfini, mais d'une certaine classe, ici `O3`. Nous avons vu dans le chapitre 4 que cela a pour effet de créer un lien de dépendance entre les classes `O2` et `O3`. Il s'agit là d'une possible manière de déclencher l'exécution de messages en cascade, comme illustré par le code Java présenté ci-après.

En Java

```
class O3 {
   private int c;
   public O3(int initC) {
    c = initC;
   }
   public void incrementeC() {
    c++;
   }
   public void afficheC() {
     System.out.println("l'attribut c est egal a: " + c);
   }
}
class O2 {
  public void jeTravaillePourO2(O3 lienO3) {
    lienO3.incrementeC();
    lienO3.afficheC();
  }
}
public class O1 {
  private O2 lienO2;
  private void jeTravaillePourO1() {
    O3 unO3 = new O3(6);
    lienO2 = new O2();
    lienO2.jeTravaillePourO2(unO3);
    unO3.afficheC();
  }
  public static void main(String[] args) {
    O1 unO1 = new O1();
    unO1.jeTravaillePourO1();
  }
}
```

Résultat

```
l'attribut c est égal à : 7
l'attribut c est égal à : 7
```

Que se passe-t-il dans ce code ? De nouveau, lors de son exécution, la méthode `jeTravaillePourO2()` recevra comme argument une copie de la valeur stockée dans le référent `unO3`. Mais comme cette valeur est, en réalité, l'adresse physique du même objet que celui créé et référé par `unO3` dans la méthode `jeTravaillePourO1()`, un second référent sera créé, qui permettra d'accéder à ce même objet. Au contraire de ce qui se passait dans

le cas précédent, la méthode `jeTravaillePour02()` affectera maintenant réellement l'objet, dont l'adresse lui sera transmise par argument.

On affecte, de ce fait, toujours un seul et même objet, et non plus une copie de celui-ci. Alors qu'il s'agit toujours du même passage par valeur, dupliquant l'original dans une zone temporaire, la variable affectée, finalement, ne sera pas qu'une copie de l'entier original dans le cas d'un argument de type prédéfini, mais bien l'objet original dans le cas présent. En fait, ce qui rend possible cette manipulation, c'est le mécanisme d'adressage indirect, qui permet à certaines variables de pointer, non pas directement sur leur valeur, mais vers une variable intermédiaire, pointant, elle, sur cette valeur.

Comme indiqué à la figure 6-1, pendant toute la durée de l'exécution de la méthode `jeTravaillePour02()`, l'objet `un03` sera référé deux fois, puis une seule fois à la fin de l'exécution de la méthode, puis, plus du tout, à la fin de l'exécution de la méthode `jeTravaillePour01()`. L'objet `un03`, à l'issue de l'exécution de ces deux méthodes, sera comme « perdu et satellisé » dans la mémoire, à la merci du ramasse-miettes (que nous découvrirons dans le chapitre 9). Comme nous l'avons vu précédemment, la possibilité pour un objet d'être référé un grand nombre de fois est inhérente à la pratique de la programmation orientée objet et sera reconsidérée, lorsque nous nous pencherons avec tristesse sur le passage de vie à trépas des objets.

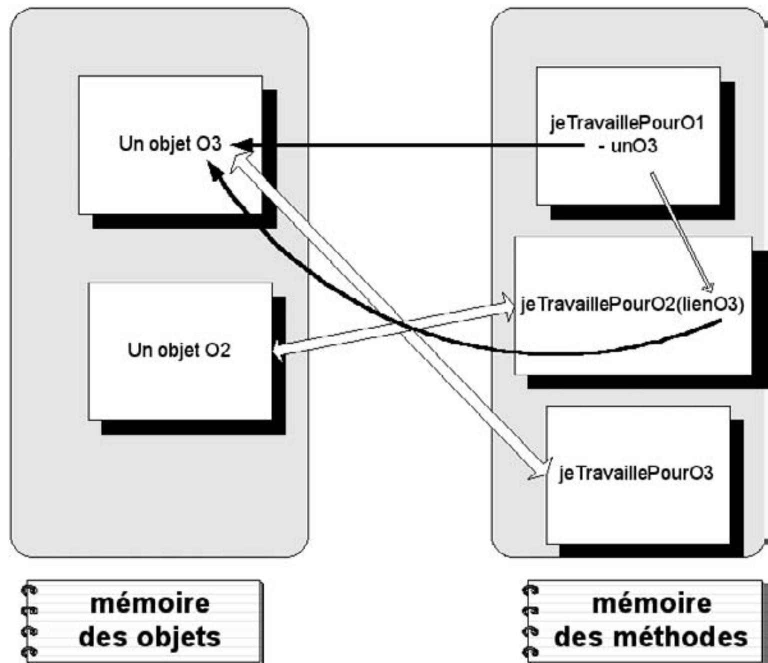


Figure 6-1
Illustration de l'effet du passage du référent « lienO3 » adressant l'objet unO3, dans la méthode jeTravaillePourO2() agissant, elle, sur l'objet O2.

En C#

En C#, la pratique et le résultat sont à première vue très semblables. Il n'y aura en général plus lieu de préciser dans la méthode que le passage se fait par référent, car, lorsque ce sont des objets qui sont passés comme argument, il n'y a simplement pas moyen de faire autrement, ils le sont par défaut. En effet, l'esprit de la programmation OO favorise ce type de passage. Ce sont bien toujours les objets originaux qui subissent des

transformations. Comme dans la vie réelle, on imagine mal qu'à chaque modification de l'état d'un objet, il faille le dupliquer afin que la modification n'affecte que la copie. Combien de copies inutiles seraient ainsi créées. Même si ces copies ne vivent que le temps d'exécution de la méthode, pendant ce temps-là, elles n'en consomment pas moins de la mémoire. Et pour quoi ? Pour rien ! Car c'est bien l'objet original que l'on cherche à transformer, et non pas cette évanescente copie parasite. Cependant, et comme le code ci-dessous l'indique, le mot-clé `ref` reste encore d'utilisation possible, y compris lors du passage des arguments référents d'objet et son emploi crée une couche additionelle d'indirection dans l'adressage.

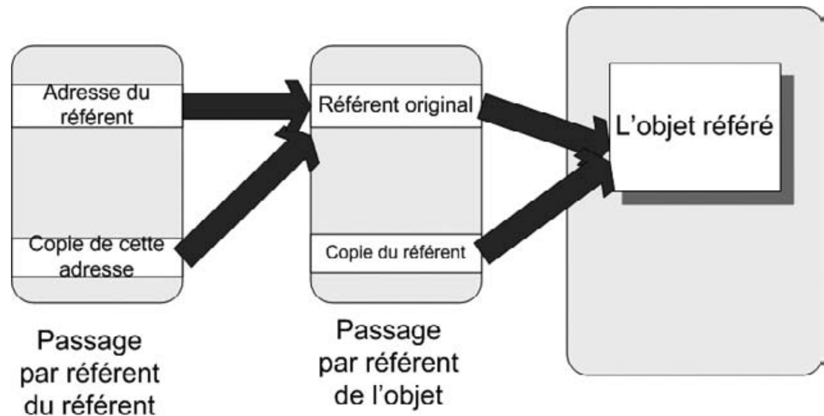
```
using System;
using System.Collections.Generic;
using System.Text;

class O3
{
    private int c;
    public O3(int initC)
    {
        c = initC;
    }
    public void incrementeC()
    {
        c++;
    }
    public void afficheC()
    {
        Console.WriteLine("l'attribut c est égal à: " + c);
    }
}

class O2
{
    public void jeTravaillePourO2(O3 lienO3)
    {
        lienO3.incrementeC();
        lienO3.afficheC();
    }

    public void jeCreeUnObjetO3(ref O3 lienO3) //le résultat sera différent selon que l'on utilise
    ➥ou pas ref
    {
        lienO3 = new O3(7);
        lienO3.incrementeC();
        lienO3.afficheC();
    }

    public static void Main()
    {
        O3 unO3 = new O3(6);
        O2 unO2 = new O2();
        unO2.jeTravaillePourO2(unO3);
```

```
        un03.afficheC();
        un02.jeCreeUnObjet03(ref un03);
        un03.afficheC();
    }
}

Résultat
l’attribut c est égal à : 7
l’attribut c est égal à : 7
l’attribut c est égal à : 8
l’attribut c est égal à : 7 ou 8 // dépendant du passage par référent ou non
```

Dans ce code, la méthode `jeCreeUnObjet03` se comportera différement selon que le passage du référent se fasse, à son tour, par référent ou par valeur. Si le passage du référent se fait par référent, et comme l’illustre la figure 6-2, on se retrouve avec un double niveau d’adressage indirect (et c’est le moment idéal pour vous mettre sur la tête et adopter pendant quelques minutes votre position de méditation zen favorite). Si le référent est passé par référent, c’est bien le référent original que vous affectez au nouvel objet à l’intérieur de la méthode et non plus sa copie, avec pour effet que le nouvel objet créé se trouvera référé par le référent de départ, laissant l’objet référé originellement perdu dans la mémoire et à la merci du « garbage collector ». Bien que tout cela soit correct sur le plan grammatical, cet extrait fait plus ressembler ce livre à un manisfeste dadaïste qu’à un livre de programmation.

Figure 6-2

Les deux niveaux d’adressage indirect découlant de la double utilisation des référents. D’abord c’est l’adresse de l’objet qui est dédoublée, ensuite c’est l’adresse de cette dernière qui l’est.



En PHP 5

Le code PHP 5 ci-après est une copie parfaite du code C#. Une des grandes innovations du PHP 5 par rapport à ses versions précédentes est, justement, d’avoir rendu le passage d’arguments objet automatiquement par référent (avant il l’était automatiquement par valeur comme en C++) sauf à le spécifier différemment, par l’addition du &.

```
<html>
<head>
<title> Passage d'arguments </title>
</head>
<body>
```

```
<h1> Passage d'arguments </h1>
<br>

<?php
   class O3 {
        private $c;

             public function __construct($initC) {
                          $this->c = $initC;
             }

             public function incrementeC(){
                          $this->c++;
             }

             public function afficheC() {
                          print ("l'attribut c est egal a $this->c <br> \n");
             }
   }

   class O2 {

             public function jeTravaillePourO2(O3 $lienO3){
                          $lienO3->incrementeC();
                          $lienO3->afficheC();
             }

             public function jeCreeUnObjetO3(O3 &$lienO3){/* avec ou sans
                                                 le & */
                          $lienO3 = new O3(7);
                          $lienO3->incrementeC();
                          $lienO3->afficheC();
             }
   }

   $unO3 = new O3(6);
   $unO2 = new O2();
   $unO2->jeTravaillePourO2($unO3);
   $unO3->afficheC();
   $unO2->jeCreeUnObjetO3($unO3);
   $unO3->afficheC();

?>
</body>
</html>
```

En C++

C++, à la différence de C# et de Java, vous oblige à traiter les arguments objets, tout comme vous traiteriez n'importe quelle variable. C'est là encore un lourd tribut payé à son funeste géniteur, le C. Aucun traitement de faveur pour les objets ! C'est bien normal pour un langage qui ne se voulait pas objet au départ. Il faudra donc préciser, comme dans le code ci-après, lors de la définition de la méthode, si le passage de l'objet `O3` se fait par valeur ou par référent.

```
#include "iostream.h"
class O3 {
private:
  int c;
public:
   O3(int initC){
    c = initC;
   }
   void incrementeC() {
    c++;
   }
   void afficheC() {
     cout <<"l'attribut c est egal a: " << c << endl;
   }
};
class O2 {
public:
/*
  void jeTravaillePourO2(O3 lienO3)  {
    lienO3.incrementeC();
    lienO3.afficheC();
  }
*/
  void jeTravaillePourO2(O3  &lienO3) {
    lienO3.incrementeC();
    lienO3.afficheC();
  }
};
class O1 {
private:
  O2 *lienO2;
public:
  void jeTravaillePourO1() {
    O3 unO3(6);
    lienO2 = new O2();
    lienO2->jeTravaillePourO2(unO3); /* appelle de manière semblable la première version
    ➥ ou la seconde*/
    unO3.afficheC();
  }
};
int main() {
    O1 unO1;
    unO1.jeTravaillePourO1();
    return 0;
}
```

Dans ce code, la première méthode reçoit l'objet comme valeur et la seconde comme référent. Les deux ne peuvent être utilisées simultanément, car leur appel se passe de la même manière. Il faut donc lever cette ambiguïté, et choisir l'une ou l'autre de ces manières. Selon que l'on utilise la version par valeur (première méthode) ou par référent, le résultat sera différent.

Résultat passage par valeur

```
l'attribut c est égal à : 7
l'attribut c est égal à : 6
```

Résultat passage par référent

```
l'attribut c est égal à : 7
l'attribut c est égal à : 7
```

On constate que, dans le premier cas, l'objet original passé comme argument est laissé inchangé (seule la copie a été affectée) alors que, dans le second, c'est bien l'objet original qui a été modifié.

En Python

Python, à nouveau, comme Java et comme C#, et comme le code ci-dessous l'indique, lorsqu'il s'agit de référents sur les objets, transmet bien l'adresse en argument et donc, indirectement l'objet original qui se trouvera modifié par l'exécution de la méthode.

```
class O3:
        __c=0
        def __init__(self, initC):
                self.__c=initC
        def incrementeC(self):
                self.__c+=1
        def afficheC(self):
                print "l'attribut c est egal à: %s" %(self.__c)

class O2:
        def jeTravaillePourO2(self,lienO3):
                lienO3.incrementeC()
                lienO3.afficheC()


class O1:
        __lienO2=0
        def jeTravaillePourO1(self):
                unO3=O3(6)
                lienO2=O2()
                lienO2.jeTravaillePourO2(unO3)
                unO3.afficheC()
unO1=O1()
unO1.jeTravaillePourO1()
```

Résultat

```
l'attribut c est égal à : 7
l'attribut c est égal à : 7
```

Comme le passage des objets se fait, par défaut, par valeur en C++, de nombreux objets seront soumis à des clonages temporaires, qu'il faudra réaliser avec soin. Nous verrons au chapitre 9 que, ce clonage demandant une attention toute particulière, C++ vous invite à utiliser un constructeur particulier, appelé « constructeur par copie », qui entre en action dès qu'un objet est cloné.

> **Passage par référent**
>
> La programmation orientée objet favorise dans sa pratique le passage des arguments objets comme référent plutôt que comme valeur. C'est toujours sur ces mêmes malheureux objets que la majorité des méthodes s'acharnent sans créer de nouveaux cobayes le temps de leurs méfaits. Les langages Java, C#, PHP 5 et Python en ont fait, légitimement, leur mode de fonctionnement par défaut, alors que le C++ s'est limité à généraliser aux objets le passage par valeur propre aux variables de type prédéfini. Une des lourdeurs inhérentes au C++ est qu'il faudra recourir à une pratique non intuitive (due à l'utilisation explicite de pointeurs ou de référents) pour obtenir le comportement, *a priori*, le plus intuitif.

Une méthode est-elle d'office un message ?

Nous avons vu que message il y a quand une méthode intervient dans l'interaction entre deux objets. Les concepts de message et de méthode deviennent-ils dès lors synonymiques ? Pas vraiment et ce pour plusieurs raisons. Le message ramène la méthode à sa seule signature. Pour qu'un objet s'adresse à un autre, il doit uniquement connaître la signature de la méthode, et peut se désintéresser complètement du corps de cette dernière. Ce qu'il doit connaître de la méthode, c'est son mode d'appel, c'est-à-dire : son nom, ses arguments et le type de ce que la méthode retourne, pour autant qu'elle retourne quelque chose.

Même message, plusieurs méthodes

Le fait de tenir la signature séparée du corps de la méthode permet aussi de prévoir plusieurs implémentations possibles pour un même message, implémentations qui pourraient, ou évoluer dans le temps, sans que le message lui-même ne s'en trouve affecté, ou, toujours plus fort, qui pourraient différer, selon la nature ultime de l'objet à qui le message est destiné. Dans un film des Monty Python, au départ d'un 100 m pour coureurs qui n'ont pas le sens de l'orientation, le même coup de feu déclenchait le départ des coureurs dans toutes les directions. Au contraire, lors de la même épreuve pour sourds, le coup de feu laissait tous les coureurs de marbre. C'est d'ailleurs de ces comiques que provient le nom d'un des langages de programmation que nous utilisons dans ce livre. On vous laisse deviner lequel… Nous verrons dans les chapitres 12 et 13 que cette variation sur un même thème est permise en OO : elle est nommée « polymorphisme ».

> **Message = signature de méthode disponible**
>
> Le message se limite uniquement à la signature de la méthode : le type de ce qu'elle retourne, son nom et ses arguments. En aucun cas, l'expéditeur n'a besoin, lors de l'écriture de son appel, de connaître son implémentation ou son corps d'instructions. Cela simplifie la conception et stabilise l'évolution du programme.

Interface : liste de signatures de méthodes disponibles

Toutes les signatures de méthodes ne deviendront pas des messages pour autant. Nous expliquerons dans les deux chapitres suivants la pratique de « l'encapsulation », qui n'octroie qu'à un nombre restreint de méthodes l'heureux privilège de pouvoir être appelées de l'extérieur. Pour l'instant, vous pouvez vous borner à lier ce terme à la seule utilisation des objets limonade, bière ou Bacardi. L'idée est de pouvoir extraire de la définition de chaque classe la liste des signatures de méthode qui pourront faire l'objet d'envoi de messages. De manière quelque peu anticipée, nous appellerons cette liste l'interface de la classe, car il s'agit bien de la partie visible de la classe, seule disponible pour des utilisateurs extérieurs. Dans la figure 6-3, vous observerez l'extraction, à partir de la définition des classes, des seules méthodes qui pourront faire l'objet de messages.

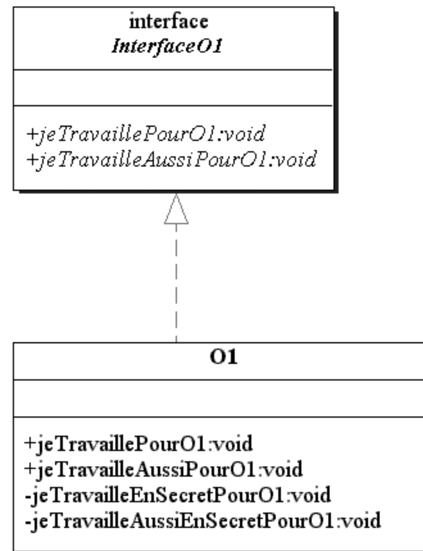


Figure 6-3

Extraction de l'interface de la classe O1, ne reprenant que les signatures des méthodes disponibles pour les autres classes.

Nous préciserons aussi, plus avant dans cet ouvrage, au chapitre 15, le lien entre la classe `O1` et son interface, dénommée ici `InterfaceO1`. Pour l'instant, le seul point à retenir est que chaque objet se caractérisera par une liste de messages disponibles, son interface, que d'autres pourront déclencher sur lui. Dorénavant, ce sera l'interface, plus que la classe directement, qui reprendra les services que l'objet sera en mesure de rendre à tout autre. Les objets sont d'une pudeur extrême et ne montre que leur interface aux autres objets. Pourquoi extraire de la classe cette seule partie visible ? Car il n'est pas nécessaire pour un premier objet, utilisant les méthodes du second, d'avoir accès à toutes ces méthodes, surtout si celles-ci risquent d'évoluer au cours du temps. Certaines relèvent du fonctionnement interne et intime de l'objet, et ne peuvent être actionnées que par l'objet lui-même.

Des méthodes strictement intimes

Quand le conducteur démarre sa voiture, il change de vitesse puis appuie sur la pédale d'accélération. Il se moque éperdument de savoir que, lorsqu'il appuie sur cette pédale, il accroît la dimension de l'entrée du mélange gazeux dans le moteur. Il laisse le soin à la pédale elle-même de communiquer avec le moteur, de manière à prolonger le seul service que le conducteur exige de sa voiture : accélérer.

Quand nous sauvegardons un fichier, nous laissons le soin au traitement de texte de stocker de manière fiable tout ce qui est écrit sur le disque dur. Il devra repérer un espace libre sur le disque, éventuellement fractionner le fichier en un ensemble de morceaux qu'il devra se préoccuper de relier entre eux, et associer, également sur le disque, le nom du fichier à l'adresse où celui-ci se trouve. Pour ce faire, le traitement de texte, lui-même, utilisera les services du pilote du disque dur intégrés dans le système d'exploitation. En fait, un des rôles majeurs de tout système d'exploitation informatique est de fournir à l'utilisateur ou à toute application qui le requiert un ensemble d'interfaces, « amicales » (traduction littérale du fameux « user-friendly » américain, dans une conception de l'amitié très éloignée de celle que Montaigne vouait à La Boétie), pour réaliser très intuitivement un ensemble de services, dont l'implémentation complexe, invisible à vos yeux et à ceux de l'application, est entièrement laissée au soin du système d'exploitation lui-même.

> **Interface**
> La liste des messages disponibles dans une classe sera appelée l'interface de cette classe.

La mondialisation des messages

Message sur Internet

Un message se limite-t-il à circuler dans la mémoire vive de l'ordinateur, comme nous l'avons vu dans les chapitres précédents, ou peut-il franchir les murs, les frontières, les océans, les planètes et les univers… Oui, il peut franchir tout cela, et bien plus encore, pour autant qu'il trouve là-bas un objet à qui s'adresser, et qui a prévu, là-bas, de par sa classe, de pouvoir répondre à ce message. Il existe une manière qui s'est extraordinairement répandue aujourd'hui pour relier des objets informatiques entre eux… On vous la donne en mille… Eh oui ! Internet. Deux objets pourront se parler à travers Internet, non pas pour s'envoyer par e-mail des plaisanteries salaces ou des spams qui ne le sont pas moins, mais pour se charger mutuellement de certains services, occupation bien plus noble, s'il en est…

Pour qu'un premier objet parle à un second, il lui sera maintenant important de connaître, non seulement son nom, mais également son adresse Internet, de manière à retrouver l'ordinateur sur lequel cet objet s'activera. Il lui faudra bien évidemment posséder l'interface des services rendus par ce distant interlocuteur. Tout aussi important, il s'agira également de définir une stratégie d'activation de l'objet destinataire. Par exemple, faudra-t-il exécuter l'application qui active l'objet, avant que celui-ci ne soit réquisitionné pour exécuter son message, ou l'objet sera-t-il automatiquement activé, dès que l'ordinateur qui peut l'exécuter recevra le message ?

Quelques complications apparaîtront, par rapport au simple envoi de message dans un seul et même ordinateur. Cependant, l'ambition des concepteurs des mécanismes d'objets distribués (Java-RMI, CORBA ou services web) est de rendre l'aspect Internet le plus transparent possible, c'est-à-dire, qu'à quelques détails près, vite assimilés, comme l'adresse des objets et les stratégies d'activation, la réalisation d'applications distribuées soit en tout point semblable à celle d'applications locales.

L'informatique distribuée

En fait, tout le mécanisme de communication entre objets par envois de messages a permis de repenser la conception des applications informatiques distribuées. La distribution d'applications informatiques à travers un réseau reste le fait qu'un programme s'exécutant sur un ordinateur puisse, à un certain moment, déléguer une partie de sa tâche à un autre programme, s'exécutant sur un autre ordinateur. Cela se faisait déjà. Simplement, tout a été repensé et reformulé à la sauce OO. Ce ne sont plus des procédures qui s'appellent à distance, mais des objets qui se parlent à distance, par envoi de messages. Le chapitre 16 sera entièrement dédié aux objets distribués.

> **Les objets distribués**
> Les technologies d'objets distribués tentent d'étendre à tout Internet la portée des envois de message entre objets, et ce de la manière la plus simple et transparente qui soit.

Exercices

Exercice 6.1

Qu'afficheront à l'exécution les deux programmes suivants ? Notez l'obligation pour la méthode `main`, statique, de ne pouvoir intégrer dans son code que des attributs ou des méthodes également déclarés statiques.

Code : Chapitre6.java

```
public class Chapitre6 {
  static int i;
  static public void test(int i) {
    i++;
    System.out.println ("i = " + i);
  }
  public static void main(String[] args) {
    i = 5;
    test(i);
    System.out.println("i = " + i);
  }
}
```

Code : Chapitre6.csc

```
using System;
public class Chapitre6 {
  static int i;
  static public void test(ref int i) {
    i++;
    Console.WriteLine ("i = " + i);
  }
  static public void test(int i) {
    i++;
    Console.WriteLine ("i = " + i);
  }
  public static void Main() {
    i = 5;
    test(i);
    Console.WriteLine("i = " + i);
    test(ref i);
    Console.WriteLine("i = " + i);
  }
}
```

Exercice 6.2

Qu'affichera à l'exécution le code C++ suivant ?

```
#include "stdafx.h"
#include "iostream.h"
void test(int i) {
  i++;
  cout <<"i = "<<i<<endl;
}
```

```
void test2(int &i) {
  i++;
  cout <<"i = "<<i<<endl;
}
int main() {
  int i = 5;
  test(i);
  cout <<"i = "<<i<<endl;
  test2(i);
  cout <<"i = "<<i<<endl;
  return 0;
}
```

Exercice 6.3

Qu'affichera à l'exécution le code Java suivant ?

```
class TestI {
  int i;
  public TestI(int i) {
    this.i = i;
  }
  public void getI(){
    System.out.println ("i = " + i);
  }
  public void incrementeI(int i) {
    this.i += i; // this est le référent de l'objet lui-même
  }
}
public class Chapitre6 {
  static TestI unTest = new TestI(5);
  static int i = 5;
  static int j = 5;

  static void test(int j) {
     i+=j;
     j+=5;
  }
  static void Test(TestI unTest) {
    unTest.incrementeI(i);
    unTest.incrementeI(j);
  }
  public static void main(String[] args) {
    test(i);
    Test(unTest);
    unTest.getI();
  }
}
```

Que donnerait ce même code si nous remplacions dans la méthode `static void test(int j)` le nom de l'argument par `k`, ainsi que si l'instruction de la méthode, `i+=j`, devenait `i+=k` ?

Exercice 6.4

Qu'affichera à l'exécution le code C++ suivant ?

```
#include "stdafx.h"
#include "iostream.h"
class TestI {
private:
  int i;
public:
  TestI(int i) {
    this->i = i;
  }
  void getI() {
    cout << "i = " << i << endl;
  }
  void incrementeI(int i) {
    this->i += i; // this est le référent de l'objet lui-même
  }
};
void test(int i) {
  i++;
  cout <<"i = "<<i<<endl;
}
void Test(TestI unTest, int i) {
    unTest.incrementeI(i);
}
void Test2(TestI &unTest, int i) {
    unTest.incrementeI(i);
}
int main() {
  int i = 5;
  test(i);
  TestI unTest(5);
  TestI *unAutreTest = new TestI(5);

  Test(unTest, i);
  unTest.getI();
  Test2(unTest, i);
  unTest.getI();

  Test(*unAutreTest, i);
  unAutreTest->getI();
  Test2(*unAutreTest, i);
  unAutreTest->getI();

  return 0;
}
```

Exercice 6.5

Qu’affichera à l’exécution le code Java présenté ci-après ?

```
class TestI {
  int i;
  public TestI(int i) {
    this.i = i;
  }
  public void getI() {
    System.out.println ("i = " + i);
  }
  public void incrementeI(int i) {
    this.i += i; // this est le référent de l'objet lui-même
  }
}
public class Chapitre6 {
  static TestI unTest = new TestI(5);
  static void Test(TestI unTest) {
    unTest = new TestI(6);
    unTest.incrementeI(5);
  }
  public static void main(String[] args) {
    Test(unTest);
    unTest.getI();
  }
}
```

Exercice 6.6

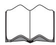
Pourquoi C# ne permet le passage par « référent » que pour des arguments de type prédéfinis ? En quoi C++ ne choisit pas la facilité en utilisant par défaut le passage d’arguments par valeur pour les objets ?

7

L'encapsulation des attributs

Ce chapitre a pour objet d'introduire la pratique d'encapsulation que, dans un premier temps, nous limiterons aux seuls attributs. Cette encapsulation pour les attributs est justifiée par la préservation de l'intégrité des objets, la lecture des attributs détachée du stockage et la stabilisation des codes.

Sommaire : Private ou public — Attributs private — Encapsulation — L'intégrité des objets — Gestion d'exception — Stabilisation des codes



Candidus — Certains termes de l'OO, tels « encapsulation » et « cloisonnement », me font penser à hermétisme et réglementation. Nous n'allons pas embrigader notre pauvre bébé tout de même ! Les nouveaux langages me semblaient pourtant offrir plus de souplesse ! Pourquoi donc ce « touche pas à ma classe ? »

Doctus — Je ne vois aucune contradiction entre souplesse et élégance. L'encapsulation est le moyen de ranger proprement le contenu de chaque objet. Il faut y voir un souci de répartition des tâches.

Cand. — Proprement ?

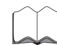
Doc. — Oui, chaque objet doit traiter l'information d'une manière qui lui soit propre.

Cand. — Et concrètement, j'y gagnerai quoi ?

Doc. — En tout premier lieu, les accesseurs (*setters* et *getters*) doivent faire partie de l'interface des objets. Cette interface est rigoureusement spécifiée. Il en résulte que les éléments internes, variables et méthodes, pourront évoluer sans la moindre conséquence pour les objets environnants. Ce qui nous sera bien utile lorsque le fabricant décidera de changer quelque chose à l'intérieur de ses jouets. Et tu constateras qu'on n'y perd pas en liberté au bout du compte. Cette encapsulation n'est pas autre chose qu'un emballage. Il permet d'éviter les fuites…

Cand. — … ou les mains baladeuses. Au lieu de me servir dans les affaires de quelqu'un, mieux vaut donc lui demander poliment. Il saura toujours mieux que quiconque où il les a rangées.

Doc. — Il saura également faire mieux que toi le nécessaire pour s'assurer que tu peux en disposer en toute sécurité. Par exemple, il te fera attendre en cas de besoin.

Cand. — Ces objets font encore mieux que simplement communiquer, ils coopèrent, en fait ! 

La charte du bon programmeur OO : Arthur J. Riel

Souvent dans notre ouvrage, nous faisons référence à une supposée charte de la bonne programmation OO. Cette charte, en fait, n'existe que dans notre imagination. Ce n'est autre qu'une compilation d'un ensemble de bonnes pratiques OO, acquises naturellement après tant d'années passées devant l'écran, et disséminées çà et là dans la grande quantité d'ouvrages que nous avons lus, avant d'oser nous lancer tête baissée dans le nôtre. Néanmoins, il est un écrit qui pourrait presque prétendre à ce titre : nous l'avons rencontré sous la plume de Arthur J. Riel, sous le titre *Object-Oriented Design Heuristics* (Addison-Wesley). Plus de 60 recommandations de bonnes pratiques OO y sont présentées, illustrées et défendues. En voici quelques-unes, piochées au hasard, et que vous rencontrerez pour certaines plusieurs fois dans ce livre :

- Minimisez le nombre de messages dans le protocole d'une classe.
- N'encombrez pas la partie publique d'une classe avec des choses que les utilisateurs de cette classe ne sont pas aptes à utiliser ou dont ils ne voient pas l'intérêt.
- Une classe capture une et une seule abstraction.
- Ne créez pas de classes « God » dans votre application. Soyez très sceptiques devant toute classe s'appelant en partie « Pilote », « Manager », « Système » ou « Sous-Système ».
- Méfiez-vous des classes contenant beaucoup de méthodes d'accès. Cela impliquerait que les données et l'utilisation que l'on en fait ne sont pas tenues dans une seule et même classe.
- Minimisez le nombre de classes avec lesquelles une autre classe collabore.
- Minimisez le nombre de messages qui peuvent être envoyés entre une classe et celles qui collaborent avec cette dernière.
- L'héritage ne devrait être utilisé que pour modéliser une relation de spécialisation.
- Les superclasses ne doivent rien savoir de leurs sous-classes.
- N'utilisez pas le mot-clé« protected ».
- L'héritage devrait être très profond ; plus il l'est, mieux c'est.
- Toutes les superclasses devraient être abstraites.
- Factorisez les attributs, les méthodes, aussi haut que possible dans la hiérarchie d'héritage.

Certaines de ces recommandations sont parfois discutables, et toute bonne pratique souffre toujours d'un point de vue très subjectif de la chose. Toutefois, l'effort est plus que louable et, indéniablement, cette tentative d'énoncer toutes ces exhortations les unes à la suite des autres dans un seul livre aura très positivement impressionné la communauté informatique. Celle-ci y a répondu largement, en faisant de certaines d'entre elles des préceptes presque aussi incontournables pour les programmeurs que ne le furent les dix commandements d'un certain Moïse pour le peuple hébreu.

Accès aux attributs d'un objet

Accès externe aux attributs

Lorsque, dans notre écosystème du chapitre 3, l'objet `proie` boit l'eau, et dès le moment où la proie possède parmi ses attributs un possible accès à l'objet `eau` (par la présence d'un référent), pourquoi ne pourrait-elle pas directement s'occuper de la diminution de la quantité d'eau, sans ce détour obligé par une méthode de la classe `Eau` qui s'en charge elle-même ? En tous les cas, il serait plus facile d'écrire directement :

```
class Proie {
  Eau eau
  void bois() {
      eau.quantite = eau.quantite - 1000; // plutôt que eau.diminueQuantite(1000)
  }
}
```

que de passer par la méthode de l'eau `diminueQuantite()`, rajoutée à cet effet dans la classe `Eau`, et qui se limite à refaire exactement la même chose, c'est-à-dire diminuer la quantité d'eau. De même, pourquoi le `feu-de-signalisation`, quand il passe au vert, ne pourrait-il pas directement changer la vitesse de la voiture, à l'aide d'une instruction telle que `laVoitureDevant.vitesse = 50`, sans devoir passer, là encore, par une méthode de la classe `Voiture` ? En fait, pratiquement tous les langages de programmation OO, dans la lignée du C++, le permettent, à tort comme nous le verrons, pour autant que l'on déclare explicitement les attributs comme `public`. Et nous voici en présence d'un nouveau mot-clé, capital de la programmation OO, qui caractérise l'accès aux attributs et aux méthodes de la classe par toute autre classe. Ce mot-clé ne devrait idéalement prendre que deux valeurs : `public` et `private` (laissons pour l'instant les deux mots en anglais, vu qu'ils le sont dans les langages de programmation, en exprimant nos regrets les plus sincères auprès de la French Academy).

Attribut *private*

Un attribut ou une méthode sera *private*, si l'on souhaite restreindre son accès à la seule classe dans laquelle il est déclaré. Il sera *public* si son accès est possible par, ou dans, toute autre classe.

Cachez ces attributs que je ne saurais voir

Si vous nous avez suivi jusqu'ici, permettez-nous de vous poser la question suivante : comment, jusqu'à présent et de façon implicite (nous n'avons pour l'instant encore jamais fait allusion au mode d'accès), avons-nous considéré le mode d'accès des attributs d'une classe : `private` ou `public` ? Ouf ! vous nous avez fait peur… Mais oui, `private`, bien entendu ! Il est inconcevable que vous ayez pu répondre autre chose. Nous avons dit et redit dans les chapitres précédents que les seuls accès possibles aux attributs d'une classe, y compris leur simple lecture, ne pouvaient se faire que par l'entremise des méthodes de cette classe.

En déclarant les attributs comme `private`, toute tentative d'accès direct, du genre : `o2.unAttribut = 50` (quand dans l'objet `o1`, la valeur de l'attribut `unAttribut` de l'objet `o2` se voit directement changée), sera verbalisée par le compilateur. Ce mot-clé, `private` ou `public`, permet au compilateur de nous seconder (toujours ce même côté cerbère dans les langages qui en font un usage bien entendu), en faisant d'une mauvaise pratique orientée objet une erreur de syntaxe (et de compilation). Dans le petit code suivant, on a rajouté le mode d'accès à la déclaration des attributs et des méthodes, rendant maintenant complète la déclaration de notre classe.

```
class Feu-de-signalisation {
  private int couleur ; /* attribut à l'accès privé */
  private Voiture voitureDevant ;  /* autre attribut de type référent à l'accès privé */

  public Feu-de-Signalisation (int couleurInit, Voiture voitureInit) { /* le constructeur sera presque
  ➥ toujours public évidemment, puisqu'on crée un objet de l'extérieur de la classe de cet objet */
    couleur = couleurInit ;
    voitureDevant = voitureInit ;
   }
  public void change()  /* une autre méthode accessible de l'extérieur */ {
    couleur = couleur + 1 ;
    if (couleur == 4) couleur = 1 ;
    if (couleur  == 1) voitureDevant.changeVitesse(50) ;
  }
}
```

Encapsulation des attributs

Sachez que certains langages OO, et non des moindres – car ce sont de vénérables langages du troisième âge (40 ans en informatique), comme Smalltalk –, rendent impossible, pour les attributs, tout autre accès que `private`. Tous les attributs seront `private` par défaut, circulez, y a rien à voir ! Dans les langages plus modernes et moins scrupuleux, dès le moment où, respectant ainsi la charte de la bonne programmation OO, vous déclarez explicitement ces attributs `private`, leur simple lecture ou modification se fera, à l'aide de méthodes d'accès, comme dans les cinq codes en cinq langages qui suivent. Dans ces codes, la classe `FeuDeSignalisation` est déclarée avec le mode d'accès des attributs adéquats. Ensuite, un objet issu de cette classe est créé et son attribut couleur reçoit la valeur 1.

En Java

```
class FeuDeSignalisation {
  private int couleur;  /* l'attribut privé */
  public FeuDeSignalisation(int couleur)  /* le constructeur presque d'office public */ {
    if ((couleur > 0) && (couleur <= 3))
      this.couleur = couleur;
  }
  public int getCouleur()  /* la méthode qui renvoie la valeur de la couleur */ {
    return couleur;
  }
  public void setCouleur(int nouvelleCouleur) /* une méthode qui modifie la valeur de la couleur */ {
    if ((nouvelleCouleur > 0) && (nouvelleCouleur <= 3))
      couleur = nouvelleCouleur;
  }
}
public class Principale {
  public static void main(String[] args) {
    FeuDeSignalisation unFeu = new FeuDeSignalisation(2);
    System.out.println(unFeu.getCouleur()); /* on affiche la valeur de la couleur */
    unFeu.setCouleur(1); /* on modifie cette valeur */
    /* unFeu.couleur = 1  est une instruction interdite */
  }
}
```

En C++

```
#include "stdafx.h"
#include "iostream.h"
class FeuDeSignalisation {
private : /* on factorise le private et le public */
  int couleur;
public:
  FeuDeSignalisation(int couleur) {
    if ((couleur > 0) && (couleur <= 3))
      this->couleur = couleur;
  }
  int getCouleur() {
    return couleur;
```

```
  }
  void setCouleur(int nouvelleCouleur) {
    if ((nouvelleCouleur > 0) && (nouvelleCouleur <= 3))
      couleur = nouvelleCouleur;
  }
};
int main() {
    FeuDeSignalisation unFeu(2);
    cout << unFeu.getCouleur() << endl;
    unFeu.setCouleur(1);
    return 0;
}
```

La seule différence sensible à relever avec Java est qu’il n’est pas nécessaire de répéter le mot-clé `public` ou `private`, quand, plusieurs attibuts ou méthodes à la suite, partagent un même mode d’accès.

En C#

```
using System;

class FeuDeSignalisation {
  private int couleur;
  public FeuDeSignalisation(int couleur) {
    if ((couleur > 0) && (couleur <= 3))
      this.couleur = couleur;
  }
  public int accesCouleur /* méthode d’accès très originale */ {
    get {
      return couleur;
    }
    set {
      if ((nouvelleCouleur > 0) && (nouvelleCouleur <= 3))
        couleur = value;
    }
  }
}
public class Principale {
  public static void Main() {
    FeuDeSignalisation unFeu = new FeuDeSignalisation(2);
    Console.WriteLine(unFeu.accesCouleur);
    unFeu.accesCouleur = 1;
   }
}
```

En C#, les modes d’accès, `set` et `get`, sont regroupés dans une seule méthode. `value` indique la valeur à transmettre dans l’attribut. L’appel de la méthode se fait tout comme un accès direct à un attribut quelconque. Comme il s’agit, en effet, d’une forme « indirecte » d’accès à l’attribut, on conçoit mieux l’existence de cette syntaxe.

En PHP 5

```
<html>
<head>
<title> Encapsulationn </title>
</head>
<body>
<h1> Encapsulation </h1>
<br>

<?php
   class FeuDeSignalisation {
            private $couleur;

            public function __construct($couleur) {
                     if (($couleur > 0) && ($couleur <=3))
                              $this->couleur = $couleur;
            }

            public function getCouleur() {
                     return $this->couleur;
            }

            public function setCouleur($nouvelleCouleur) {
                     if (($nouvelleCouleur > 0) && ($nouvelleCouleur <=3))
                              $this->couleur = $nouvelleCouleur;
            }
   }

   $unFeu = new FeuDeSignalisation(2);
   print($unFeu->getCouleur());
   $unFeu->setCouleur(1);
   $unFeu->couleur = 3; /* Le programme se plante ici
            et ne fait plus rien*/
   print($unFeu->getCouleur());
?>

</body>
</html>
```

Rien de bien spécial à dire. Bien évidemment, en l’absence de compilation, c’est lors de l’éxécution qu’une tentative d’accès à quoi que ce soit de privé dans la classe déclenchera une erreur fatale.

En Python

```
class FeuDeSignalisation:
      __couleur = 0
      def __init__(self,couleur):
              if couleur>0 and couleur <=3:
                      self.__couleur=couleur
      def getCouleur(self):
              return self.__couleur
      def setCouleur(self,nouvelleCouleur):
              if nouvelleCouleur>0 and nouvelleCouleur<=3:
                      self.__couleur=nouvelleCouleur
```

```
unFeu=FeuDeSignalisation(2)
print unFeu.getCouleur()
unFeu.setCouleur(1) #changement de notre attribut privé
unFeu.__couleur = 2 #cela n'affecte en rien l'attribut privé
                  #un nouvel attribut est simplement créé
print unFeu.getCouleur() #valeur de notre attribut privé
print unFeu.__couleur #valeur du nouvel attribut
```

Résultat

```
2
1
2
```

En Python, comme le code l’illustre, la mise en œuvre des attributs privés est encore différente, dû à l’absence d’étape de compilation préalable. Pas de mot-clé `private`, un attribut privé est simplement signalé par la présence de deux underscores pour précéder son nom. Lorsque la déclaration de celui-ci est rencontrée à l’exécution, son nom est automatiquement changé de manière invisible (ces changements s’opérant également là où il apparaît dans les méthodes), ce qui fait que toute tentative d’accès par la suite ne concerne plus ce même attribut. L’attribut en devient inaccessible en dehors des méthodes de la classe.

Mais pourquoi ces mille détours avant de lire ou de modifier un attribut ? Pourquoi les classes ne peuvent-elles exhiber leurs attributs en public ? Il y a plusieurs justifications à l’obligation, morale nous l’avons vu (car il y a possibilité de contourner cette obligation), de déclarer les attributs comme `private`. Nous retrouverons certaines de ces justifications, lorsque nous discuterons du mode d’accès des méthodes, qu’il faut également privilégier comme `private`. Bien sûr, tout ne peut être privé dans ce club très sélect que sont les classes, car on n’y verrait plus grand monde. Pas tout, mais beaucoup de choses néanmoins.

Encapsulation

L’encapsulation est ce mécanisme syntaxique qui consiste à déclarer comme `private` une large partie des caractéristiques de la classe, tous les attributs et de nombreuses méthodes.

Encapsulation : pourquoi faire ?

Pour préserver l’intégrité des objets

Et tout d’abord pourquoi sommes-nous instamment priés de déclarer les attributs comme `private` ? Une première raison étend la responsabilité des classes, non seulement au typage de leurs objets, mais également à la préservation de l’intégrité de ces derniers. En général, les objets d’une classe, décrits et caractérisés par la valeur de leurs attributs, ne peuvent admettre que ces attributs prennent toute et n’importe quelle valeur. La couleur du feu ne peut prendre que les valeurs 1, 2 et 3. La quantité d’eau ne peut devenir négative, de même que l’énergie des animaux. Pourtant, rien dans la déclaration même des attributs ne permet ces restrictions. Vous allez dire que tout programmeur est suffisamment malin et prévoyant pour deviner ce que l’on fera de ces attributs (ces attributs et non les siens). Le programmeur de la classe elle-même, sans doute, mais pourquoi les programmeurs de toutes les autres classes, appelées à interagir avec la première, devraient-ils également se préoccuper de cette intégrité ?

Rendons à chaque programmeur la classe qui lui appartient. Mieux vaut prévenir que guérir… Un coup de paille lors de la compilation d’un programme est préférable à un coup de poutre à son exécution (ou quelque chose du genre…). Laissons ainsi, à chaque classe, le soin de s’assurer qu’aucun de ses objets ne subira de changements d’état non admis. L’unique manière de procéder consiste à rendre les attributs inaccessibles, sinon par l’entremise de méthodes publiques, c’est-à-dire accessibles, elles, et qui s’assureront que les nouvelles valeurs prises restent dans celles admises. Charité bien ordonnée commence par soi-même (c’est le dernier dicton, promis !).

Vous comprendrez très facilement comment les méthodes peuvent s’en charger, en lisant les deux petits codes qui suivent.

```
class Feu-de-signalisation {
  private int couleur;
  private Voiture voitureDevant;

  public void changeCouleur(int nouvelleCouleur) {
    if (nouvelleCouleur >= 1) && (nouvelleCouleur <=3) /* intégrité assurée */
      couleur = nouvelleCouleur ;
  }
}
class Voiture {
    private int vitesse ;
    public int changeVitesse(int nouvelleVitesse) {
      if (nouvelleVitesse >= 0) && (nouvelleVitesse <=130) /* intégrité assurée */
        vitesse = nouvelleVitesse ;
      return vitesse ;
    }
}
```

En quelque sorte, les méthodes de la classe filtrent l’usage que l’on fait des attributs de la classe. En entrée, elles ne toléreront que certaines valeurs. En sortie, elles présenteront les attributs, d’une manière qui convient aux autres classes, à celles qui veulent connaître leur valeur. C’est ce que Betrand Meyer tente d’installer d’une manière moins forcée dans sa version des langages OO, par l’introduction des notions d’invariance, par le fait qu’un objet puisse, avant d’éxécuter une méthode, vérifier qu’un ensemble de pré-conditions soit satisfait et, qu’à l’issue de cette exécution, c’est un ensemble de post-conditions qui le soit.

Intégrité des objets

Une première raison justifiant l’encapsulation des attributs dans la classe est d’obliger cette dernière, par l’intermédiaire de ses méthodes, à se charger de préserver l’intégrité de tous ses objets.

La gestion d’exception

Dans nos cinq langages OO, il est également prévu que toute tentative, lors de l’exécution du programme, visant à violer l’intégrité d’un objet, en lui passant des valeurs d’attributs inadmissibles, puisse faire l’objet d’un mécanisme de gestion d’exception. Ce mécanisme permet, soit à la classe elle-même, soit à son interlocutrice, de prévoir et de prendre en compte la réponse à donner à cette tentative avortée : on interrompt le programme, la classe interlocutrice essaie une autre valeur, on continue comme si de rien n’était, mais, cette fois-ci, sans avoir à effectuer le changement. La gestion d’exception est un mécanisme de programmation assez sophistiqué, destiné à la réalisation de code plus robuste et qui permet d’anticiper et de gérer les problèmes

pouvant survenir lors de l’exécution d’un code dans un contexte sur lequel le programmeur n’a pas tout contrôle. Il pourrait faire l’objet d’un chapitre à lui tout seul.

Toute source de problèmes pouvant survenir à l’exécution n’est pas évitable, tel un réseau ou un disque dur inaccessible, un processeur inapte au multithreading, un accès incorrect à une base de données, un entier devant servir de dividente égal à zéro et beaucoup d’autres. En général, les instructions susceptibles de poser de tels problèmes sont placées dans un bloc `try-catch`. Lorsque le problème se pose effectivement, le programmeur est censé l’avoir anticipé et avoir prévu dans la partie catch du bloc une manière de récupérer la situation, un filet de sûreté, afin de reprendre le code à ce stade. Sans cela, le code s’interrompt en déclenchant juste l’exception. En présence du `try-catch`, le code continue et exécute le remède que le programmeur a prévu en réponse à ce problème. Un ensemble d’exceptions déjà répertoriées (comme le fameux `NullPointerException` en Java ») existent dans les librairies associées aux différents langages de programmation et ne demandent alors qu’à être simplement « rattrapées ». En héritant de la classe `Exception` (comme dans le code ci-après), le programmeur peut créer ses propres classes d’exception, en accord avec la logique de son code et de manière à bénéficier de ce mécanisme de gestion d’exceptions prêt-à-l’emploi.

Dans le code Java qui suit, nous nous limitons à en montrer un exemple à titre pédagogique, dans lequel le programmeur du `FeuDeSignalisation` prévoit à l’avance ce qui devra se produire si un quelconque utilisateur du code tente de changer la couleur du feu en lui passant une valeur non autorisée.

Exemple Java d’exception

```
class FeuDeSignalisation {
  private int couleur;

  public void changeCouleur(int nouvelleCouleur) throws MauvaiseCouleurException {
   if ((nouvelleCouleur >= 1) && (nouvelleCouleur <=3)) /* intégrité assurée */
    couleur = nouvelleCouleur ;
   else throw new MauvaiseCouleurException(nouvelleCouleur);
   /* C’est à cet endroit précis du code qu’on génère l’exception pour des couleurs
      non autorisées */
 }
}
/* Puis on définit la classe, sous-classe d’exception qui indiquera ce qu’il y a lieu de faire */
class MauvaiseCouleurException extends Exception {
  public MauvaiseCouleurException(int couleur) {
   System.out.println("La couleur " + couleur + " que vous avez rentree n'est pas permise");
 }
}

public class TestException {
  public static void main(String[] args) {
   FeuDeSignalisation unFeu = new FeuDeSignalisation();
    try { // Toute exception doit être intégrée dans un bloc “ try - catch ”
         unFeu.changeCouleur(5);
  }
    catch (MauvaiseCouleurException e) {System.out.println("L'exception s'est declenchee");}
 }
}
```

Résultats

```
La couleur 5 que vous avez rentree n'est pas permise
L'exception s'est declenchee
```

La gestion d'exception

Toujours dans la perspective de sécuriser au maximum l'exécution des codes, tous les langages OO que nous présentons intègrent dans leur syntaxe un mécanisme de gestion d'exception dont la pratique est très voisine. Seul change le recours obligatoire ou non à cette gestion, Java étant le plus contraignant en la matière. En Java, la non-prise en compte de l'exception sera signalée et interdite par le compilateur. Une exception est levée quand quelque chose d'imprévu se passe dans le programme. Il est possible alors « d'attraper (try-catch) » cette exception et de prendre une mesure correctrice qui permette de continuer le programme malgré cet événement inat-tendu. Paradoxalement, toute la gestion d'exception consiste à rendre l'inattendu plus attendu qu'il n'y paraît, et de se préparer au maximum à toutes les éventualités problématiques ainsi qu'à la manière de les affronter. À l'instar de Java, il apparaît donc assez cohérent de forcer le programmeur à en faire usage.

Pour cloisonner leur traitement

Une deuxième justification à l'encapsulation des attributs, et partagée avec l'encapsulation des méthodes, comme nous le verrons dans le chapitre suivant, est de renforcer la stabilité du logiciel à travers le temps et ses multiples et possibles évolutions. Nous avons vu que la classe permet une décomposition naturelle du logiciel, en autant de modules à répartir entre plusieurs programmeurs. Afin que les travaux de chacun des programmeurs ne doivent faire l'objet de révision et d'adaptation, à chaque changement par l'un d'entre eux d'une partie de son code, il est extrêmement important de rendre les codes les plus indépendants possible entre eux. Il faut limiter l'impact dans le reste du code d'un quelconque changement dans une petite partie de ce dernier.

Autorisant tous les modules fonctionnels à interagir avec l'ensemble des données du problème, la programmation procédurale ne favorise en rien cette stabilité. En effet, toute transformation dans le typage ou le stockage des données affectera tous ces modules. Comme, de surcroît, ces modules s'imbriquent entre eux, l'impact se propagera, tant en largeur qu'en profondeur. En programmation OO, en revanche, toute modification d'une partie `private` de la classe n'aura aucun impact sur le reste du programme. Il est bien connu par les développeurs de logiciel que la maintenance du code constitue une dépense aussi importante, sinon plus importante, que l'obtention d'une première version. De manière à diminuer cette dépense, il est capital qu'un travail d'anticipation, concrétisé par l'encapsulation, entraîne les programmeurs à séparer, dans le développement de leur classe, ce qui restera stable dans le temps (en le déclarant comme `public`) de ce qui est susceptible, encore, de possibles modifications (en le déclarant comme `private`).

Pour pouvoir faire évoluer leur traitement en douceur

Les attributs et leur typage sont de façon typique une partie de code susceptible de nombreuses évolutions dans le temps. Déjà, la manière même de sauvegarder l'état de l'objet sur le disque dur, dont nous traiterons au chapitre 19, risque d'être revue à travers le temps : sauvegarde en tant qu'objet, sauvegarde séparée des attributs dans un fichier ASCII, sauvegarde en tant qu'enregistrement d'une base de données relationnelle, sauvegarde dans une base de données orientée objet. Il devient alors capital, afin de neutraliser l'impact d'un tel changement, de déclarer comme `private` tout ce qui concerne le stockage des attributs. En effet, seul le type de la lecture de l'attribut, et nullement la manière de le stocker ou le coder, devrait concerner toute autre classe désirant y avoir accès.

Considérons la petite situation suivante, qui n’ira pas sans rappeler un certain « bogue » devenu tristement célèbre. À chaque objet `voiture` est rajouté un attribut codant la date de fabrication que, dans un premier temps, nous décidons, idiotement d’accord (mais c’est uniquement pour l’exemple !), de coder en tant qu’entier écrit sur 8 chiffres, par exemple 20120415 pour le 15 avril 2012 (quatre chiffres pour l’année, c’est cher mais plus malin), et de déclarer cet attribut comme `public` (plus si malin que ça !). La classe `Voiture` sera codée de la manière suivante :

```
class Voiture {
  public int dateFabrication ;
  //…… autres attributs ……
  // ……  autres méthodes……
}
```

Considérons également une autre classe, modélisant les possibles acheteurs de véhicule qui, dans les différentes méthodes qui les caractérisent telles que : `calculPrix()`, `négociePrix()`, `comparePrixAvecArgus()`, `achete()`, font souvent référence à la date de fabrication de la voiture. Par exemple, la méthode `négociePrix()` pourrait se définir comme suit, en tolérant un accès direct à la date de la voiture :

```
class Acheteur {
  private Voiture voitureInteressante ;

  public int négociePrix() {
    int prixPropose = 0 ;
     if (voitureInteressante.dateFabrication < 19970101) /* accès possible à l’attribut date */
      prixPropose = voitureInteressante.getPrixDeBase() - 10000;
  }
}
```

Supposons maintenant que le programmeur de la classe `Voiture` se rende compte, après quelques mois, de l’incongruité qu’il y a à coder la date de cette manière et décide, plus logiquement, de la coder comme un `String`, c’est-à-dire une chaîne de caractères, par exemple : « 15/04/2012 ». Automatiquement, l’instruction conditionnelle `if (voitureInteressante.dateFabrication < 0101997)` devient complètement absurde et provoque l’ire du compilateur, car une chaîne de caractères ne peut se comparer à un entier. Le pauvre programmeur de la classe `Acheteur` en sera réduit à entièrement récrire le code de sa classe (ne le plaignons pas, plus d’un programmeur s’étant enrichi de la sorte), vu qu’il y a de fortes chances que la date de fabrication des voitures soit souvent reprise dans ce code.

Quelle solution aurait-elle été plus sécurisée, en garantissant plus de résistance aux changements (nous voulons des programmeurs progressistes mais des classes conservatrices) ? Il aurait fallu que le programmeur de la classe `Voiture` anticipe que l’attribut `dateFabrication` puisse subir de nombreux changements dans le temps, et décide qu’il devienne adéquat, dès lors, de séparer son stockage de sa lecture. Dorénavant, quelle que soit la manière dont cet attribut sera typé et stocké, manière déclarée *`private`*, il sera toujours lu, donc présenté aux autres classes, comme un `String`. Prévoir que, dans dix mille ans d’ici, une date sera toujours conçue comme une chaîne de caractères n’est pas faire preuve de si grand don de prescience.

La bonne version du code de la classe `Voiture` devient :

```
class Voiture {
  private int dateFabrication ;
  //…. autres attributs ….
```

```
  public String getDateFabrication() {
    String date = null;
    // ... instructions qui transforme l'entier
    //date en un string ……;
    return date ;
  }
  // …… autres méthodes  ….
}
```

Cette nouvelle écriture de la classe conduira à accroître la stabilité de l'ensemble du logiciel car, si le stockage ou le typage de l'attribut `dateFabrication` change, il faudra simplement adapter le corps d'instructions de la méthode de la classe `Voiture` qui renvoie l'attribut. Aucune autre classe ne se trouvera plus affectée et, de ce fait, l'impact d'un tel changement restera confiné à la classe elle-même.

La classe : enceinte de confinement

En plus d'un type, d'un fichier, d'un garant de l'intégrité des objets, la classe se doit d'être, également, une enceinte de confinement. La conception du logiciel demande un travail d'anticipation, destiné à ne laisser *public* que ce qui est appelé à se transformer le moins, au fil du temps et des versions du logiciel. Il est évident que cette pratique ne prend vraiment toute sa raison d'être qu'avec le grossissement des projets informatiques et la multiplication des programmeurs. Plus la taille d'un programme devient importante, plus il est crucial de pouvoir facilement le décomposer et de distribuer les modules entre plusieurs développeurs, qui seront incités, dans leur pratique, à rechercher l'équilibre parfait entre les modifications incessantes de leur code et le peu d'impact que celles-ci provoquent sur les développements de leurs collègues. C'est aussi la raison pourquoi ce même mécanisme est souvent difficile à faire avaler aux étudiants qui, pour l'essentiel de leurs travaux de programmation, réaliseront, seul ou à très peu, un minuscule programme de mille lignes, sur lequel ils maintiendront l'entièreté du contrôle et qu'ils se dépêcheront d'oublier une fois l'évaluation obtenue. Situations parfaitement antagonistes à celles qui réclament l'encapsulation. Plus d'un étudiant a été surpris à déclarer les attributs comme public... Ah ! les traîtres…

> **Stabilisation des développements**
>
> Il y a une seconde raison de déclarer les attributs comme `private`, commune aux méthodes : c'est d'éviter que tout changement dans le typage ou le stockage de ceux-ci ne se répercute sur les autres classes.

Exercices

Exercice 7.1

Réalisez un petit code qui stocke un attribut `date` comme un entier, et autorise sa lecture par les autres classes uniquement comme un `String`.

Exercice 7.2

Si une classe contient 10 attributs, combien de méthodes d'accès à ses attributs vous paraissent-elles nécessaires ?

Exercice 7.3

Réalisez une classe de type `compte en banque`, en y intégrant deux méthodes, l’une déposant de l’argent, l’autre en retirant, et dont vous vous assurerez que l’attribut `solde` ne puisse jamais être négatif.

Exercice 7.4

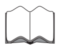
Dans la même classe que celle de l’exercice précédent, écrivez une méthode d’accès au solde, qui retourne ce dernier comme un entier, alors qu’il est stocké comme un réel.

8

Les classes et leur jardin secret

Ce chapitre poursuit l'exposé de la pratique de l'encapsulation en l'étendant aux méthodes. Il sépare l'interface d'une classe de son implémentation. Il justifie cette encapsulation par la stabilisation des développements qu'elle améliore. Il discute les différents niveaux d'encapsulation rendus possibles dans les langages de programmation. Il termine par une petite allusion aux systèmes complexes dont se rapproche tant la pratique de l'OO et qui contribue à en permettre la maîtrise.

Sommaire : Méthode publique ou privée — Interface et implémentation — Améliorer la stabilité des développements — Niveaux intermédiaires d'encapsulation : amitié, héritage, paquetage, classes imbriquées — L'effet papillon dans les systèmes complexes



Doctus — Imagine que nous souhaitions modifier un mécanisme interne à un objet. Nous en avons toute liberté pour peu qu'on ait pris la précaution de limiter son usage, à celui privé de notre boîte noire !

Candidus — Oui, mais la modification des méthodes publiques qui font partie de l'interface perturbe les relations avec les autres objets du programme !

Doc. — Ce n'est pas dit… Tu peux fournir un accès public à une fonctionnalité tout en évitant d'en dire trop. Il s'agit juste de dire à un objet ce qu'on attend de lui sans pour autant lui dire comment il doit s'y prendre ! C'est lui qui doit savoir comment implémenter la chose, avec ses propres méthodes privées.

Cand. — Et c'est comme ça que je pourrai les bidouiller sans craindre d'entendre crier : « Ça marche plus, je parie que t'as encore fais une modif au message faismoica_302_v1r3() ! ».

Doc. — L'héritage lui-même doit être réglementé, les parents doivent pouvoir décider de ce qu'ils gardent pour eux.

Cand. — Je pourrais donc appliquer ce même principe de cloisonnement face aux utilisateurs de mes classes ! Ils n'hériteront que des méthodes que j'aurai choisi de mettre à leur disposition.

Doc. — On peut également constituer des relations de groupe. Des classes d'objets travaillant en équipe pourront utiliser un vocabulaire commun tout en restant inaccessibles au public.

Cand. — Ni privé ni public, un jargon de spécialistes en quelque sorte !

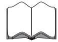
Doc. — Ou comme des amis qui parlent de ce qu'ils ont en commun alors que l'entourage n'est pas du tout concerné. Encore plus fort : dans une voiture, le volant, l'accélérateur et le frein peuvent n'exister que pour « l'objet » conducteur.

Cand. — Hmmm… Ton conducteur me fait penser à un chauffeur esclave de sa voiture qui attend ses ordres pour pouvoir s'amuser sur ses pédales.

Doc. — On peut effectivement faire dans le genre poupées russes : des objets complètement imbriqués les uns dans les autres.

Cand. — Des classes d'objets privées alors !

Doc. — Autre direction maintenant. Après la fermeture de nos boîtes noires pour interdire l'accès à leurs rouages internes, il faudra également prévoir de les interconnecter pour construire notre système. Il nous restera à doser raisonnablement la complexité des branchements du réseau de communications entre tous ces objets. Il s'agira alors d'éviter les cascades d'événements inextricables !



Encapsulation des méthodes

Idéalement, même la simple lecture des attributs ne devrait que très rarement constituer le contenu de méthode publique. La raison en est simple. Les autres classes ont-elles jamais besoin de simplement lire les attributs d'une classe donnée ? Exceptionnellement. Le plus souvent, elles modifient ces attributs ou les utilisent à travers une méthode, afin qu'à partir de la valeur de ceux-ci, une nouvelle activité se déclenche, quitte à se propager de classes en classes. Simplement les lire, et rien d'autre, n'apparaîtra que très rarement utile. Cela nous amène naturellement à une nouvelle règle de bonne conduite OO, à rajouter à la charte du bon artisan OO :

Méthode *private*

En plus des attributs, une bonne partie des méthodes d'une classe doit être déclarée comme `private`.

Interface et implémentation

On différencie les méthodes *private* des méthodes *public* en déclarant, comme nous l'avions anticipé dans un chapitre précédent, que les premières sont responsables de l'implémentation de la classe, alors que les secondes le sont de l'interface de la classe. Comme indiqué à la figure 8-1, la partie interface d'une classe doit rester réduite par rapport à son implémentation. Plus cette partie sera réduite, plus la possibilité d'un changement dans celle-ci est réduite, et moins conséquent devient le possible impact des changements dans cette classe. Gardez à l'esprit que l'interface ne reprend de toutes les méthodes de la classe, que les seuls possibles messages, c'est-à-dire les signatures des méthodes publiques. Ce sont ces seules signatures qui ne peuvent évoluer dans le temps, car même le corps des méthodes identifiées par ces signatures peut être modifié, sans conséquence sur les autres classes. De son côté, la partie private est un large espace maintenu de modifications possibles, tout comme un chantier en cours.

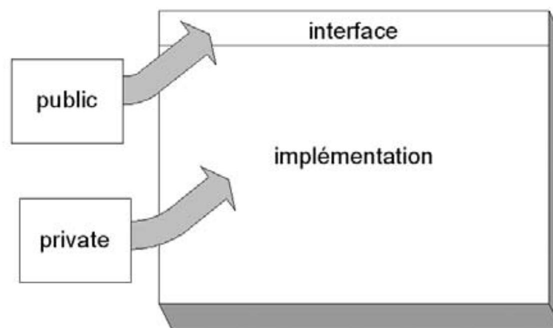


Figure 8-1

La séparation dans une classe entre une large partie implémentation et une plus petite partie interface.

Toujours un souci de stabilité

Cette séparation force tout programmeur d'une classe à réfléchir de façon anticipée, afin de tenir clairement détachées les méthodes qu'ils prédestinent aux autres classes de celles qui font partie du jardin secret de la classe qu'il programme. Tout changement dans une classe qui se produit dans les méthodes d'implémentation n'affectera d'aucune manière le codage de toutes les classes interagissant avec celle-là. Les méthodes `private` de la classe agissent dans la mesure où elles sont appelées dans les méthodes `public` de cette classe. Ce qui se révèle ne pas être possible pour les premières, c'est qu'elles soient appelées de l'extérieur de la classe. Ainsi dans les deux petits codes qui suivent, en Java (en C++, C# et PHP c'est parfaitement équivalent) et en Python, la méthode privée `pasSetCouleur()`, qui se déclenche si la couleur passée n'est pas autorisée, ne pourra être appelée que de l'intérieur de la classe.

En Java

```
class FeuDeSignalisation {
  private int couleur;

  public FeuDeSignalisation(int couleur) {
     if ((couleur >= 1) && (couleur <=3)) {
       this.couleur = couleur ;
      }
  }

  public int getCouleur() {
    return couleur;
  }

  private void pasSetCouleur(int nouvelleCouleur) {
     System.out.println ("pas bonne couleur, la: " + nouvelleCouleur);
  }

  public void setCouleur(int nouvelleCouleur) {
    if ((nouvelleCouleur >= 1) && (nouvelleCouleur <=3))
      couleur = nouvelleCouleur ;
    else pasSetCouleur(nouvelleCouleur); // appel de la methode privée
  }
}

public class TestPrive {
    public static void main(String[] args) {
       FeuDeSignalisation unFeu = new FeuDeSignalisation(2);
       System.out.println(unFeu.getCouleur());
       unFeu.setCouleur(5);
       System.out.println(unFeu.getCouleur());
       /* unFeu.pasSetCouleur(5); ici, on ne peut appeler cette méthode
           privée */
    }
}
```

Resultats

```
2
pas bonne couleur, la : 5
2
```

En Python

```
class FeuDeSignalisation:
        __couleur = 0
        def __init__(self,couleur):
                if couleur>0 and couleur <=3:
                     self.__couleur=couleur

        def __pasSetCouleur(self,nouvelleCouleur): #methode privée par
                                                   #le double underscore
           print "pas bonne couleur, la: %s" %(nouvelleCouleur)

        def getCouleur(self):
                return self.__couleur

        def setCouleur(self,nouvelleCouleur):
                if nouvelleCouleur>0 and nouvelleCouleur<=3:
                self.__couleur=nouvelleCouleur
                else:
                self.__pasSetCouleur(nouvelleCouleur) #appel de la méthode privée


unFeu=FeuDeSignalisation(2)
print unFeu.getCouleur()
unFeu.setCouleur(5)
print unFeu.getCouleur()
#unFeu.__pasSetCouleur(5) ici, on ne peut appeler cette méthode privée
```

Notez qu'une telle pratique, que l'on cherche à encourager par la programmation OO, est déjà monnaie courante dans bien d'autres secteurs de l'industrie. Avez-vous l'impression que l'interface des voitures, des téléphones, des frigos, des machines à laver, se soit considérablement modifiée depuis des années ? Mais conduisez aux États-Unis, et vous subirez de plein fouet les inconvénients d'un changement d'interface de la classe `Voiture` sur la classe `Conducteur` (sans parler également des contraventions pour excès de vitesse, mais cela c'est une autre histoire). En revanche, l'implémentation des moteurs ou des téléphones a subi des changements substantiels. La technologie des moteurs automobiles s'est largement améliorée, les moteurs, autrefois à injection indirecte, sont aujourd'hui à injection directe, alors que votre mode de conduite ne s'en est, en rien, ressenti. La raison, en termes OO, tient au fait que tout ce qui concerne l'allumage du mélange de carburant, l'explosion… de l'objet voiture reste du domaine privé et donc inaccessible à l'objet `conducteur`, même s'il utilise son briquet.

Les téléphones sans fil et ceux avec fil reposent sur des protocoles de communication foncièrement différents, sans que votre manière de téléphoner ne s'en trouve affectée.

Dans le premier chapitre, relatant ce que vous observiez par la fenêtre, vous vous êtes limités à ne citer que la voiture, sans détailler sa structure car, là encore, l'interface que vous utilisez ne requiert pas une connaissance

structurelle du véhicule. S'il est possible que, dans la déclaration de la classe `Voiture`, on retrouve des attributs agrégés de type moteur ou roue, il y a peu de chances que l'interface de `Voiture` y fasse une allusion explicite dans les signatures des méthodes.

Cette séparation `private/public` ne se fait pour le programmeur qu'au prix d'un travail d'anticipation concernant les fonctionnalités de ses classes qu'il juge stables dans son code pour de nombreuses années (et qu'il peut rendre publiques et accessibles) et celles qu'il soupçonne d'être susceptibles de changer (et qu'il vaut mieux garder `private`). Avez-vous constaté que lorsque vous changez l'imprimante de votre ordinateur, vous n'avez pas à recompiler toutes les applications – et elles sont nombreuses – dans lesquelles vous avez la possibilité d'imprimer un document ? C'est toujours la même idée : tous les objets imprimantes, quelle que soit la manière physique (leur implémentation) dont ils le font (laser, matriciel, jet d'encre), sont capables d'imprimer un document, et donc de s'interfacer adéquatement à la fonction `print` de ces applications. Vous ne trouverez jamais une fonctionalité de manipulation de laser dans les menus à votre disposition dans le traitement de texte que vous utilisez.

Signature d'une classe : son interface

Dans le schéma d'interaction entre classes, qui est la base de la programmation OO, il est, en vérité, plus correct de parler d'interaction entre interfaces qu'entre classes. Seules les interfaces apparaissent comme disponibles aux autres classes. De là, la pratique courante, que nous approfondirons plus avant dans le chapitre 15, qui consiste à extraire de chacune des classes la seule partie visible par les autres, celle qu'elle met à disposition des autres : son interface. À nouveau, la syntaxe de certains langages vous permet de forcer le trait (toujours cette assistance, dans les bons langages OO, de la syntaxe, à vous encourager à une bonne pratique de l'OO), par l'existence d'une structure syntaxique d'interface, qui sera héritée par la classe implémentant cette interface.

Interaction avec l'interface plutôt qu'avec la classe

Lorsqu'une classe interagit avec une autre, il est plus correct de dire qu'elle interagit avec l'interface de cette dernière. Une bonne pratique de l'OO vous incite, par ailleurs, à rendre tout cela plus clair, par l'utilisation explicite des interfaces comme médiateurs entre les classes.

Les niveaux intermédiaires d'encapsulation

Nous n'avons vu que deux modes d'accès possibles pour les propriétés d'une classe : `public`, pour les rendre accessibles à toutes les autres et qu'il convient d'utiliser avec prudence, et `private`, pour les rendre inaccessibles aux autres, et que l'on peut consommer sans modération. Certains langages de programmation introduisent des raffinements additionnels pour ce mode d'accès, en en tolérant des niveaux intermédiaires. Par exemple, une classe pourrait décider de se rendre entièrement accessible à quelques autres classes, privilégiées, qu'elle déclarera comme faisant partie de ses « amies ». Qu'une classe déclare une autre comme étant son amie (utilisation du mot-clé `friend`), et ce qui est `private` dans la première deviendra `public` pour la seconde. Elle tolérera un début d'atteinte à sa vie privée. C++ est un de ces langages qui, au contraire de Java et de C#, permettent ce raffinement additionnel dans la mise en œuvre de l'encapsulation. Ainsi, dans le petit code C++ qui suit, l'objet `O2` peut utiliser une méthode déclarée private dans la classe `O1`, car cette dernière a accepté d'ouvrir son cœur à la classe `O2`. La classe `O2` est déclarée comme `friend` de la classe `O1`.

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
  int a;
  void jeTravailleSecretementPourO1()  /* méthode déclarée private */ {
    cout <<"la valeur de a est: " << a << endl;
  }
public:
  O1(int initA):a(initA) {}
  friend class O2;  /* la classe O2 est déclarée comme amie de la classe O1, ce qui lui donne
  ➥ un droit de regard privilégié sur O1 */
};
class O2 {
public:
  O2() {
    O1 unO1(5);
    unO1.jeTravailleSecretementPourO1(); /* O2 peut utiliser cette méthode pourtant « private »
    ➥ dans O1 */
  }
};
int main(int argc, char* argv[]) {
  O2 unO2;
  return 0;
}
```

Résultat

```
la valeur de a est : 5
```

Dans les langages permettant aux classes de se faire quelques amies, comme dans la réalité hélas, l'amitié n'est ni symétrique ni transitive (non, les amis de vos amis ne seront plus automatiquement vos amis).

Une classe dans une autre

Une autre possibilité de rendre accessible les attributs et les méthodes déclarés `private` dans une classe à une autre classe se présente lorsque cette seconde classe est créée à l'intérieur de la première. Ce système de classes imbriquées l'une dans l'autre n'est pas des plus simples à mettre en œuvre, et ne devrait être exploité que très rarement, vu les autres modes, plus intuitifs, qui vous sont proposés pour associer deux classes. Les deux codes qui suivent, le premier en Java et l'autre en C#, vous montrent comment, en effet, une classe peut être déclarée à l'intérieur d'une autre. La classe englobée aura un accès privilégié à tout ce qui constitue la classe englobante. À nouveau, n'utilisez ce stratagème que si vous voulez, le plus étroitement qui soit, solidariser le fonctionnement et le développement des deux classes.

En Java

```
class O4 {
  public O4() {
    O3.DansO3 unTest = new O3.DansO3(); // il est possible d'utiliser directement la classe englobée
  }
}
```

```
public class O3  /* définition de la classe englobante */ {
  static private int a;
  public O3(int b) {
    a = b;
  }
  public static void jeTravaillePourO3() {
    a = 5;
    DansO3 unDansO3 = new DansO3();
  }
  static class DansO3  /* définition imbriquée
    d'une nouvelle classe englobée par la première */{
    private int b;
    public DansO3(){
      b = a;  /* malgré qu'il soit privé dans la classe englobante, a est accessible par la classe
      ➥ englobée */
      System.out.println("la valeur de b est : " + b);
    }
  }
  public static void main(String[] args){
    jeTravaillePourO3();
    O4 unO4 = new O4();
  }
}
```

Résultat

```
la valeur de b est : 5
la valeur de b est : 5
```

L'équivalent en C#

```
using System;
class O4 {
  public O4(){
    O3.DansO3 unTest = new O3.DansO3();
  }
}
public class O3{
  static private int a;

  public O3(int b){
    a = b;
  }
  public static void jeTravaillePourO3() {
    a = 5;
    DansO3 unDansO3 = new DansO3();
  }
  public class DansO3{
    private int b;

    public DansO3(){
      b = a;
      Console.WriteLine("la valeur de b est : " + b);
    }
```

```
    }
    public static void Main(){
      jeTravaillePour03();
      O4 unO4 = new O4();
    }
}
```

Utilisation des paquetages

Une autre possibilité encore, que nous retrouverons dans un prochain chapitre détaillant le mécanisme d'héritage, consiste à ne permettre qu'aux seuls enfants de la classe (ses héritiers) un accès aux attributs et méthodes `protected` du parent. Si j'ai droit à l'héritage, pourquoi n'aurais-je pas le droit d'exploiter toutes les caractéristiques dont j'hérite. Attendez de le savoir petit vénal ! Enfin, une ultime possibilité permet de libérer l'accès, uniquement aux classes faisant partie d'un même paquetage, en général, quand les fichiers ne contiennent qu'une classe, aux fichiers faisant partie d'un même répertoire. Par exemple, en Java, quand vous n'indiquez ni `private` ni `public`, comme mode d'accès pour les propriétés de la classe, le mode par défaut est celui limité aux seuls paquetages. Dans le code Java, ci-après, la classe `O1`, ne rend disponible sa méthode, `jeTravailleSecretementPourO1()`, précédée d'aucun mot-clé d'accès, qu'uniquement aux classes présentes dans le même paquetage ou « package » (`OO1` dans le code).

```
package OO1; /* déclaration qui intègre la classe dans le package OO1. Le nom de la classe,
➥ dorénavant, sera précédé d'OO1 */
public class O1 {
  private int a;

  void jeTravailleSecretementPourO1() /* sans rien indiquer, la méthode ne sera accessible qu'à
  ➥ partir du même package OO1 */{
    System.out.println("la valeur de a est: " + a);
  }
  public O1(int initA) {
    a = initA;
  }
}
```

Les paquetages existent également en C# et C++, où ils sont nommés `namespace`, « espace de nommage », ce qui leur correspond, en effet, plus fidèlement. L'équivalent en C# du code Java précédent est :

```
using System;
namespace OO1 { /* déclaration du namespace */
public class O1{
  private int a;

  void jeTravailleSecretementPourO1() /* sans rien indiquer ou en utilisant un mot-clé additionnel
  ➥ « internal », la méthode ne sera accessible qu'à partir du même namespace OO1 */{
    Console.WriteLine("la valeur de a est: " + a);
  }
  public O1(int initA){
    a = initA;
  }
 }
}
```

En C#, et en .Net en général, il faut néanmoins faire la différence entre les concepts de namespace et d'assembly (les .dll de Microsoft). On installe les fichiers classes dans l'assembly lors de l'opération de compilation. Par exemple, ci-dessous, les versions exécutables de trois fichiers classes sont installées dans l'assembly `exempleAssembly.dll`.

```
csc /t :library /out :exempleAssembly.dll Classe1.cs Classe2.cs Classe3.cs
```

Plutôt qu'aux `namespace`, les niveaux d'accès et d'encapsulation seront plutôt relatifs à la découpe des classes et des fichiers correspondants en « assembly ». Il y a donc tout intérêt à nommer les `namespace` et les `assembly` de la même manière, afin de faciliter la compréhension et la gestion de l'ensemble des classes. Le namespace de ces trois classes serait donc ici : `exempleAssembly`.

Ces niveaux d'accessibilité intermédiaire ont le défaut d'accroître la portée d'un changement effectué dans une petite partie du code. Ils sacrifient, de ce fait, l'effort anticipatif qui consiste à jouer de cet accès avec la plus grande prévoyance aux futurs changements plus étendus, faisant suite à une modification quelconque du code. À vous de choisir. Mais, là encore, la charte du bon programmeur OO, plébiscitée par ce livre, vous encourage à n'utiliser que les deux seuls accès, `private` et `public`, et avec une grande parcimonie quant au second.

Désolidariser les modules

Alors que les deux niveaux extrêmes de l'encapsulation – « private » : fermé à tous et « public » : ouvert à tous – sont communs à tous les langages de programmation OO, ceux-ci se différencient beaucoup par le nombre et la nature des niveaux intermédiaires. De manière générale, cette pratique de l'encapsulation permet, tout à la fois, une meilleure modularisation et une plus grande stabilisation des codes, en désolidarisant autant que faire se peut les réalisations des différents modules.

Afin d'éviter l'effet papillon

La physique d'aujourd'hui s'intéresse de près à la modélisation et la compréhension de systèmes complexes composés de multiples agents simples en interaction. Stuart Kauffman est un de ces chercheurs qui se sont appliqués à comprendre le fonctionnement de tels réseaux, et surtout, l'impact sur ce fonctionnement de la façon dont les agents sont interconnectés. De ces nombreuses études effectuées sur des systèmes et réseaux aussi variés que les réseaux de neurones, réseaux génétiques, immunitaires, verre de spin et autres, il résulte que ces systèmes ont les comportements les plus riches quand les agents ne sont pas insuffisamment connectés entre eux et quand ils ne le sont pas trop.

Stuart A. Kauffman et Albert-Lazlo Barabasi

Stuart Kauffman est un de ces chercheurs qui auront marqué (et continuent à le faire) très durablement les sciences de la complexité. Biologiste de formation, et longtemps un des piliers du célèbre Institut Santa Fe, il a depuis fondé sa propre compagnie, Bios Group, dans laquelle il met ses compétences en matière de systèmes complexes au service des problèmes économiques et managériaux. Aujourd'hui, il est retourné au monde académique en acceptant une charge professorale à l'université de Calgary. Un système complexe est un système non décomposable, constitué de multiples agents, généralement au comportement individuel simple, mais interconnectés entre eux. La biologie foisonne de pareils systèmes : écosystèmes, réseaux de neurones, réseaux immunitaires, réseaux génétiques, réseaux cellulaires, etc. Ce que Kauffman s'est surtout efforcé de montrer, par des simulations informatiques aussi originales que convaincantes, c'est qu'il existe dans ces réseaux une manière pour les agents de s'interconnecter entre eux, qui rendent leur comportement, ni totalement figé, ni totalement chaotique, mais quelque part entre les deux, au bord du chaos. C'est dans cet entre-deux, dans ce régime intermédiaire, que les systèmes exhibent les comportements dynamiques les plus riches d'intérêt, en termes d'adaptabilité et de stockage d'information.

> Auteur de nombreux ouvrages importants, un de ces derniers s'intitule simplement *Chez soi dans l'Univers – La recherche des lois de l'auto-organisation*. Dans cet ouvrage, il montre que les systèmes complexes tendent spontanément et gratuitement à se structurer de manière à produire des comportements complexes émergents.
>
> Cette complexité résulte du fonctionnement collectif des unités en interaction et se produit très simplement et très naturellement.
>
> En affirmant cela, Kauffman cherche à contrer une opinion très répandue en biologie, qui consiste à attribuer toute la complexité des systèmes à la seule évolution darwinienne. Dans cette vision, des systèmes de plus en plus complexes apparaissent car survivant au fil de l'évolution la sélection darwinienne. Les simulations de Kauffman montrent que les systèmes biologiques sont capables très spontanément de comportements complexes, sans pour cela subir de pression sélectionniste. Ce qui nous intéresse le plus dans ses travaux, c'est la nécessité pour les agents constituant ces systèmes de maintenir entre eux des interactions de portée réduite. Ils le font pour une raison très proche de celle qui justifie la structure d'interaction entre objets dans les développements OO. Comme dans les écosystèmes, comme dans le cerveau, comme dans les réseaux génétiques, les agents doivent interagir avec un minimum de leurs collègues. Et ce de manière à véhiculer l'information le plus subtilement possible, ni trop ostensiblement, tout le monde influençant tout le monde, ni trop timidement, personne n'influençant personne.
>
> Dans sa suite, Albert-Lazlo Barabasi, professeur de physique à l'université américaine de Notre-Dame, a décelé dans une très large variété de réseaux informatiques, biologiques, sociaux et de transport (réseaux qu'ils a analysé à la loupe), que ceux-ci ne présentent pas une topologie aléatoire et une structure de connectivité uniforme. En revanche, il y a décelé un petit nombre de nœuds possédant un très grand nombre de connexions. Leur nombre reste très largement inférieur à celui des nœuds faiblement connectés mais se trouve être beaucoup plus important que dans un cas purement aléatoire. Ce sont ces nœuds singuliers mais stratégiques, comme autant de carrefours de ce réseau, que l'on désigne par l'appellation de « connecteur ». Leur position centrale en matière de connectivité ainsi que leur nombre plus important que par simple tirage aléatoire les rendent responsables de nombreuses propriétés et fonctions caractérisant les réseaux qui les hébergent. Les diagrammes de classe des grands codes OO tendent à vérifier cette topologie non aléatoire en présence de quelques classes centrales fortement connectées et un très grand nombre de classes qui le sont beaucoup moins.

Par exemple, parmi les trois réseaux de la figure 8-2, c'est le troisième qui présenterait le comportement le plus riche d'intérêt. Dans un réseau insuffisamment connecté, l'isolation des agents ne permet pas à des comportements émergents, c'est-à-dire innovants par rapport au seul comportement des agents, de se produire. Toute modification de l'agent reste cantonnée à lui-même, et ne produit aucun impact sur les autres. Le système est rigide, gelé, complètement décomposable, non plus uniquement lors de sa conception, ce qui est souhaitable mais, aussi, lors de son fonctionnement, ce qui l'est beaucoup moins. En revanche, dans un réseau largement interconnecté, c'est-à-dire quand un agent se trouve en moyenne connecté à plus de 3 ou 4 autres agents, le comportement devient complètement chaotique. La moindre modification sur un agent se propage sur tous les autres, au risque de produire des comportements toujours instables et imprédictibles.

L'impact qui va sans cesse s'amplifiant, bien que résultant d'une petite perturbation locale, a été métaphoriquement dénommé par les physiciens « d'effet papillon », quand le battement d'aile d'un papillon à Paris est responsable de l'arrivée d'un ouragan en Floride. Une conséquence première serait d'intensifier la chasse aux papillons dans les rues de Paris. Mais une pratique plus facile à mettre en œuvre, c'est de limiter la connectivité entre agents à 1 ou 2 voisins, ce qui permet l'émergence de nouveaux et intéressants comportements, et surtout une certaine stabilité et robustesse de ces comportements face à des perturbations locales.

Il en va de la physique des systèmes complexes comme de la programmation orientée objet, ce qui est plutôt heureux. Dans les deux cas, on simplifie un problème en le décomposant en un ensemble d'agents simples et en interaction, dont l'effet, collectif, produit les comportements souhaités. Comme pour un réseau de neurones, un ensemble d'agents stupides, mais en relation, peut produire, *in fine*, un comportement collectif et temporel complexe. Ensuite, s'il est inévitable de faire interagir les classes pour obtenir un comportement émergent

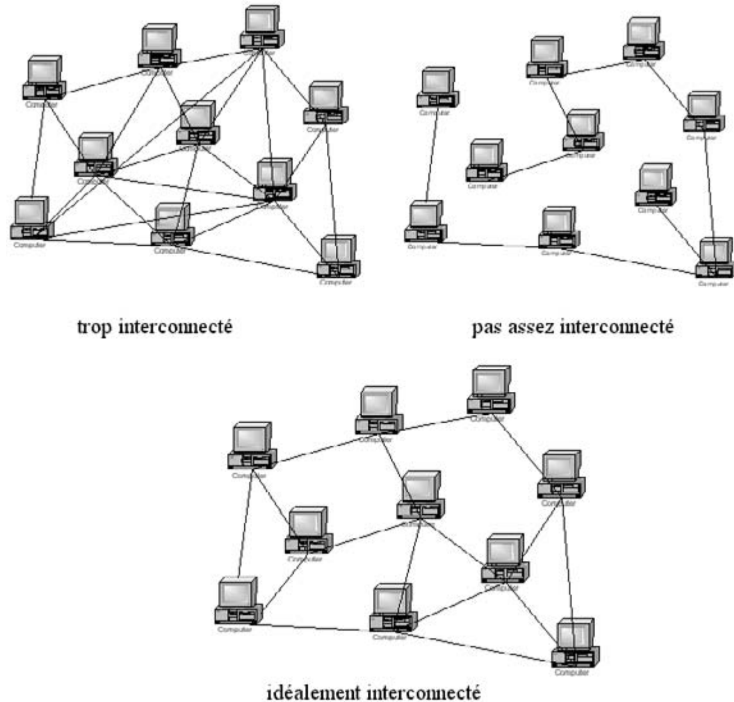


Figure 8-2
Trois structures d'interconnexion entre les éléments d'un réseau. Dans les deux premiers réseaux, les éléments sont trop ou insuffisamment connectés. Le troisième réseau présente un schéma d'interconnexion idéal.

complexe, il reste à maintenir cette interaction à un faible niveau, de manière à stabiliser l'ensemble du système face à des changements indésirables.

Il y a deux manières d'affaiblir cette interaction. La première est de ne faire interagir la majorité des classes qu'avec un minimum d'autres classes. Cela permettra de limiter la portée de l'impact d'une modification dans le code d'une classe. Par exemple, toujours dans les réseaux de neurones, chaque neurone ne se trouve connecté qu'à 1/10000000e de tous les autres. Ainsi, les espèces animales n'entretiennent des relations qu'avec très peu d'autres espèces (et les informaticiens, pour leur part, ont tendance à se reproduire entre eux…). La seconde est de ne permettre, dans chacune des classes, qu'à peu de méthodes de se transformer en messages. Il faut que l'interface soit une partie très congrue de la classe. Là encore, la portée d'un changement dans une classe s'en trouvera largement minimisée.

Certains chercheurs ont étudié la manière dont les classes étaient connectées dans les librairies Java. Ils ont remarqué un petit nombre de classes clés extrêmement connectées aux autres et un grand nombre de classes faiblement connectées. Cette structure de connectivité s'avère commune à tous les réseaux de nature humaine tels, par exemple, les réseaux d'affinité sociale : quelques connecteurs essentiels connaissent tout le monde et sont connus de tous, tandis qu'un grand nombre d'êtres humains sont beaucoup plus faiblement connectés. C'est la même structure de connectivité que l'on retrouve lorsqu'on étudie la manière dont les ordinateurs sont connectés sur Internet et dont les sites web sont connectés (par les hyperliens) sur le Web. Cette topologie allie les avantages de la robustesse (il faut éliminer les connecteurs moins nombreux pour entamer le réseau), de la vitesse de communication (ce réseau est qualifié de petit monde car les nœuds restent très proches les uns des autres) à l'économie de conception (il y a très peu de liens entre les nœuds).

Exercices

Exercice 8.1

Décrivez les différents modes d'encapsulation existant dans les langages OO et ordonnez-les, des plus sévères au moins sévères.

Exercice 8.2

Voici quelques méthodes constitutives de la classe `Voiture` ; séparez les méthodes faisant partie de l'interface de la classe de celles faisant partie de son implémentation : `tourne`, `accélère`, `allumeBougie`, `sortPiston`, `coinceRoue`, `changeVitesse`, `injecteEssenceDansCylindre`.

Exercice 8.3

Comment une méthode déclarée *private* dans une classe sera-t-elle indirectement déclenchée par une autre classe ?

Exercice 8.4

En quoi l'existence d'assemblage de classes peut compenser l'absence des relations d'amitié ?

Exercice 8.5

Pourquoi l'interface est tout ce qu'une classe `B` doit connaître d'une classe `A`, si elle désire communiquer avec cette dernière ?

Exercice 8.6

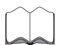
À votre avis, pourquoi l'amitié en C++ n'est-elle pas transitive ?

9

Vie et mort des objets

Ce chapitre a pour objectif de présenter les différentes manières d'effacer les objets de la mémoire pendant qu'un programme s'exécute. Nous verrons comment les langages utilisés dans ce livre traitent de ce problème : de la version libérale du C++, confiant la responsabilité au seul programmeur, aux versions plus « marxistes » de Java, Python et PHP 5, laissant un système de régulation centralisé extérieur, appelé ramasse-miettes, s'en occuper, en passant par la troisième voie chère à Tony Blair et proposée par le C#.

Sommaire : Gestion de la mémoire RAM — Dépenses de mémoire inhérentes à l'OO — Mémoire pile et mémoire tas — Le « delete » du C++ — Le ramasse-miettes de Java, C#, PHP 5 et Python



Candidus — Comment les objets s'arrangent-ils avec la mémoire ?

Doctus — Un objet est constitué d'un ensemble de données et de méthodes pour les manipuler. Lorsqu'il entre en scène un processus de chargement réserve la place nécessaire au stockage de ces deux ingrédients.

Cand. — Tu veux dire que les segments de code doivent également entrer dans les préoccupations du programmeur ! Tu parles d'un progrès !

Doc. — Bien que le code ne soit chargé qu'à un seul exemplaire, un objet est tout de même plus encombrant qu'une donnée primitive. Mais tu oublies une chose, il disposera des mécanismes nécessaires pour traiter la question. Sa suppression de la mémoire fait partie de son cycle de vie.

Cand. — Veux-tu dire que ça se fait tout seul ?

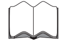
Doc. — En C++, non, mais en Java, C#, PHP 5 et Python, tu disposes de l'allocation et de la libération automatiques de mémoire.

Cand. — Je me contente donc d'appliquer les principes de localisation des données là où elles seront utilisées plutôt que de les allouer globalement au début du programme ?

Doc. — C'est bien ce que proposent ces quatre langages. Le mécanisme du ramasse-miettes, encore appelé *Garbage Collector*, prendra le soin de déterminer les circonstances où les données temporaires ont fini de servir et s'arrangera pour n'intervenir qu'en cas de réel besoin.

Cand. — Cela semble miraculeux… Comment fait ce ramasse-miettes pour savoir à coup sûr qu'une donnée ne sera plus utilisée ?

Doc. — Il n'y a aucune magie derrière tout ça ! Ces langages disposent d'un mécanisme pour détecter les occasions où le programme coupe les liens avec ses données temporaires. L'idée principale repose sur le fait que le seul moyen de créer un objet consiste à utiliser les zones mémoire contrôlées par la machine virtuelle et que cette même machine peut détecter que cet objet est devenu inutilisable et parfait pour la « casse ».



Question de mémoire

Un rappel sur la mémoire RAM

Object wanted : dead or alive ! Ce chapitre a la sinistre mais non impossible mission de vous expliquer le cycle de vie des objets, comment ils vécurent et comment ils sont morts. Vous en voulez encore ? Alors écoutez l'histoire de … Mais d'abord, quelques rappels élémentaires sur le fonctionnement d'un ordinateur lorsqu'il exécute un programme seront bienvenus en guise d'introduction. Un programme, pour qu'il s'exécute, nécessite, avant tout, de l'espace mémoire, pour pouvoir y stocker les données qu'il manipule et les instructions responsables de ces manipulations. Lors de l'exécution d'un programme OO, il faudra pouvoir stocker, et les objets et les méthodes.

La mémoire dite RAM, ou vive ou encore centrale, sert à cela. C'est une mémoire rapidement accessible, volatile et chère, au contraire du disque dur qui lui, le pauvre, est lent à la détente, mais permanent et à bon marché. De plus, elle doit se partager entre les multiples programmes qui peuvent s'exécuter en même temps, chacun ayant droit à sa part du gâteau. Les différents programmes auront une zone mémoire propre qui leur sera réservée, comme un casier dans un vestiaire, et qu'ils utiliseront exclusivement durant leur exécution. Aujourd'hui, on dit que les applications sont bien cloisonnées entre elles, ce qui permet d'éviter que l'une s'aventure dans un territoire réservé à l'autre, car, à l'instar des guerres de gangs à Los Angeles, cela peut faire beaucoup de dégâts.

Bien sûr, lorsque le programme s'interrompt, qu'il soit normalement ou anormalement terminé, toute la mémoire se vide, et c'est alors l'hécatombe du côté des objets. Et c'est bien pour cela qu'il faudra, si ce que ceux-ci sont devenus vous importe encore, vous préoccuper de sauver leur état, d'une manière ou d'une autre, sur le disque dur (nous aborderons la sauvegarde des objets sur le disque dur au chapitre 19).

Pour qu'un programme tourne vite, il est idéal que toutes les données et instructions qu'il manipule puissent être stockées dans la RAM, sinon le programme rame… Ce que l'on ne peut installer dans la RAM pourra, en dernier recours, être stocké sur le disque dur (on parle alors de mémoire virtuelle), provoquant en cela un effondrement des performances, vu que celui-ci prend pour l'extraction des données un million de fois plus de temps que la mémoire RAM. Cette mémoire-là est donc extrêmement précieuse mais, vu sa sophistication et son prix, non extensible à l'infini.

Par ailleurs, comme vous l'aurez constaté dans la pratique, en installant la nouvelle version de votre logiciel favori, plus on en a, plus on en use, pour ne pas dire abuse. La gourmandise (ou plutôt l'avidité des applications) s'adapte à la disponibilité des ressources. La mémoire RAM brûle les poches des développeurs d'application. De fait, une des préoccupations des programmeurs d'antan était d'économiser les ressources de l'ordinateur lors du développement des applications, le temps calcul et la mémoire. Aujourd'hui, l'existence même de pratique informatique comme l'OO permet de s'affranchir quelque peu de ce souci d'optimisation, pour le remplacer graduellement par un souci de simplicité, clarté, adaptabilité et facilité de maintenance. Ce que l'on gagne d'un côté, on le perd ailleurs. En effet, la pratique de l'OO ne regarde pas trop à la dépense, et ce, à plusieurs titres.

L'OO coûte cher en mémoire

L'objet, déjà en lui-même, est généralement plus coûteux en mémoire que les simples variables `int`, `char` ou `double` de type prédéfini. Il pousse à la dépense. De plus, rappelez-vous le « new », qui vous permet d'allouer de la mémoire pendant le déroulement de l'exécution du programme, et ce n'importe où. Alors pourquoi s'en priver ? À la différence d'autres langages, tout l'espace mémoire utilisé pendant l'exécution du programme n'est pas déterminé à l'avance, ni optimisé par l'étape de compilation.

Par ailleurs, certains langages OO, et non des moindres comme C++, sont des grands consommateurs d'objets temporaires utilisés, ou dans le passage d'argument ou comme variable locale (nous reviendrons sur ce point précis dans la suite). Bien que la pratique de l'OO soit une grande consommatrice de mémoire RAM, et que celle-ci va s'accroissant dans les ordinateurs suivant la fameuse loi de Moore (qui, comme un 11e commandement à force d'être citée, dit que tout en informatique fait l'objet d'un doublement de capacité tous les 16 mois), elle reste une ressource extrêmement précieuse, et toute pratique visant à économiser cette ressource pendant l'exécution du programme est plus qu'appréciable.

> **Économiser de la mémoire**
>
> La mémoire RAM est une denrée rare et chère, qu'il est important de gérer au mieux pendant l'exécution du programme, au risque de déborder sur le disque dur, avec, pour conséquence, un effondrement des performances.

Qui se ressemble s'assemble : le principe de localité

Un autre point capital dans la gestion de la mémoire est qu'il est important que les instructions et les données qui seront lues et exécutées à la suite se trouvent localisées dans une même zone mémoire. La raison en est l'existence aujourd'hui dans les ordinateurs d'un système de mémoire hiérarchisé (telle la mémoire cache), où des blocs de données et d'instructions sont extraits d'un premier niveau lent, pour être installé dans un second niveau plus rapide. Cela permet, lors de l'exécution du programme, d'extraire, le plus souvent possible, les données nécessaires à cette exécution hors du premier niveau.

Suite aux ratés, quand ce qui est requis pour la poursuite de l'exécution ne se trouve plus dans le niveau rapide, il sera nécessaire d'extirper à nouveau un bloc de données du niveau lent, en ralentissant considérablement l'exécution. Si lors du transfert de la mémoire lente vers la mémoire rapide, on ramène un peu plus que le strict nécessaire, et au vu du principe de localité, alors la probabilité d'un raté sera diminuée d'autant, car il y a de fortes chances que le surplus du transfert réponde aux prochaines requêtes. Comme les objets, au fur et à mesure de leur création, peuvent s'installer n'importe où dans la mémoire, et que l'essentiel de l'exécution consiste à passer d'un objet à l'autre, on conçoit que, là encore, la pratique OO soit presque antinomique avec toutes les démarches d'économie et d'accélération des performances. On verra qu'afin de diminuer les effets néfastes d'une telle répartition des objets, des systèmes automatiques cherchent à compacter au mieux la zone mémoire occupée par ces objets, et à les maintenir le plus possible dans la mémoire cache.

Les objets intérimaires

Si, au fur et à mesure de son exécution, le programme rajoute de nouveaux objets dans la mémoire, il serait commode, dans le même temps, de se débarrasser de ceux devenus inutiles et encombrants. Mais quand un objet devient-il inutile ? Tout d'abord, quand le rôle qu'il doit jouer est par essence temporaire. Par exemple, quand il permet à des structures de données de se transformer en passant par lui, mais n'est plus requis une fois les structures finales obtenues. En Java, existe la classe `Integer` qui permet, entre autres, de créer des

objets entiers à partir de String (chaîne de caractères), et de les manipuler, pour, une fois ces manipulations terminées, les stocker dans une simple variable de type `int`.

Dès que ce nouveau stockage est achevé, il serait intéressant de pouvoir facilement se débarrasser de l'objet `Integer`, qui a juste servi de « passerelle » entre le String et l'`int`.

Le petit code qui suit transforme l'argument String reçu lors de l'exécution du programme, par la ligne de commande indiquée ci-après, en un véritable entier correspondant. Il vous permettra également de comprendre pourquoi la méthode `main` de Java doit inclure obligatoirement un vecteur de String comme argument. Il s'agit en effet d'arguments qu'il est possible d'indiquer lors de l'exécution du programme (par exemple, le nom d'un fichier d'input…). Le passage de ces arguments se fait lors de l'instruction d'exécution du programme. Dans l'exemple, l'argument `5` est transmis comme le premier élément du vecteur de String et est ensuite transformé en l'entier « 5 ».

Ligne de commande : java ObjetInterimaire 5

```
public class ObjetInterimaire {
  public static void main(String args[]) {
    Integer unEntierInterimaire = new Integer(args[0]);
    /* On récupère le premier String passé en argument par args[0] */
    int a = unEntierInterimaire.intValue(); //transformation
    /* la méthode intValue() appliquée sur l'objet Integer permet d'en
     * récupérer la valeur entière, à ce stade-ci, l'objet
     * unEntierInterimaire n'est plus utile et pourrait être supprimé */
    System.out.println(args[0] + " s'est transforme en " + a);
  }
}
```

Résultat

```
5 s'est transforme en 5
```

Mémoire pile

Mais ce qui est vrai des objets l'est et l'a toujours été de n'importe quelle variable informatique, qui n'aurait de rôle à jouer que pendant un court laps de temps, et à l'occasion d'une fonctionnalité bien précise. Considérez le petit programme suivant, qui serait écrit de manière très semblable dans pratiquement tous les langages informatiques, dans lequel une méthode s'occupe de calculer, à partir de trois arguments reçus, les deux bases et la hauteur, la surface d'un trapèze.

Nous avons déjà abordé ce type de mécanisme dans le chapitre 6. Comme indiqué dans la figure 9-1, durant l'exécution de cette méthode, cinq variables intermédiaires vont se créer et disparaîtront aussitôt l'exécution terminée. D'abord, lors de l'appel de la méthode, trois variables nouvelles seront nécessaires pour stocker les trois dimensions du trapèze. Si ces variables existent déjà à l'extérieur de la méthode, elles seront purement et simplement dupliquées, pour être installées dans ces variables intermédiaires. Ensuite, pendant l'exécution de la méthode, une quatrième variable intermédiaire, `surface`, est créée, qui permettra de stocker le résultat jusqu'à la fin de cette exécution. Si la surface est calculable, une cinquième et dernière variable intermédiaire : `somBases`, permettra de stocker temporairement une valeur intermédiaire, dont l'usage, un peu forcé ici, permet en général une meilleure lisibilité du programme, et une algorithmique plus sûre, car décomposée en une succession d'étapes plus simples.

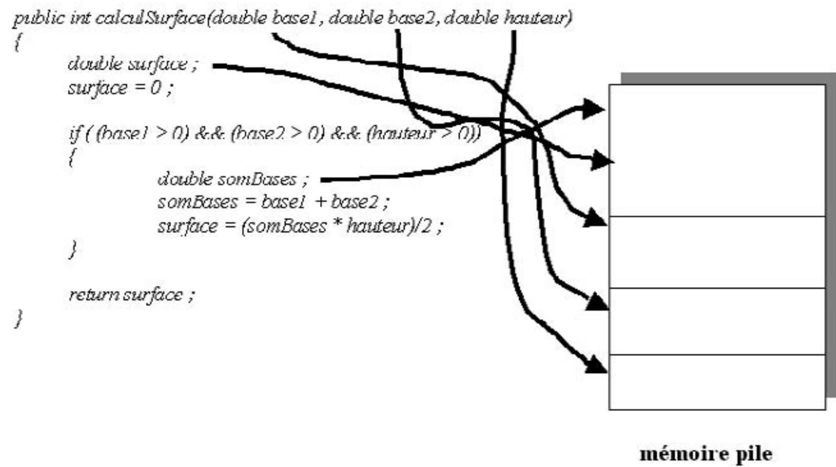


Figure 9-1

Illustration de l'existence de cinq variables temporaires utiles au calcul de la surface d'un trapèze.

Il vous paraîtra évident qu'une fois la méthode achevée, toutes ces variables doivent disparaître de la mémoire pour laisser la place à d'autres, et sans qu'on ne les y invite. C'est ce qu'elles feront dans pratiquement tous les langages, et ce le plus simplement du monde. Ces variables sont stockées, comme indiqué dans la figure, dans une mémoire dite mémoire « pile » (et qui ne s'use que si l'on s'en sert). Le principe de fonctionnement de cette mémoire est dit « LIFO » (dernier dedans premier dehors, essayez en anglais et vous comprendrez pourquoi ce fonctionnement n'a pas été dénommé « DDPD »). Dans tout code, un bloc d'instructions, encadré par les accolades, délimite également la portée des variables.

Dans une informatique séquentielle traditionnelle (nous verrons une autre solution à cela lorsque nous discuterons du « multithreading » dans le chapitre 17), un bloc d'instructions ne sera jamais interrompu. Quand un bloc se termine, les variables du dessus de la pile disparaissent tout naturellement (car elles ne sont utilisables qu'à l'intérieur de ce bloc), alors que, lorsqu'un bloc s'entame, les nouvelles variables s'installent au-dessus de la pile. Aucune recherche sophistiquée n'est nécessaire pour retrouver les variables à supprimer. Ce seront toujours les dernières à s'être installées sur la pile. De même, de cette façon, aucun gaspillage n'est possible, et aucune zone de mémoire inoccupée peut se trouver, comme un petit village gaulois, perdu au milieu de zones occupées.

Gestion par mémoire pile

Ce système de gestion de la mémoire est donc extrêmement ingénieux, car il est fondamentalement économe, gère de façon adéquate le temps de vie des variables intermédiaires par leur participation dans des fonctionnalités précises, garde rassemblées les variables qui agissent de concert, et synchronise le mécanisme d'empilement et de dépilement des variables avec l'emboîtement des méthodes.

Ce système de gestion de mémoire est très efficace pour des objets essentiellement intérimaires. Il l'est tant et si bien que C++ et C# l'ont préservé pour la gestion de la mémoire occupée par certains objets (les trois autres, quant à eux, l'ont interdit pour les objets). Idéalement, dans les deux premiers langages, vous utiliserez ce mode de gestion pour des objets dont vous connaissez à l'avance le rôle intermittent qu'ils sont appelés à jouer. Par exemple, en C++, lorsque vous créez un objet `o1` de la classe `O1`, au moyen de la simple instruction :

`O1 o1`, n'importe où dans le code, sans l'utilisation du `new` et de pointeur, vous installez d'office l'objet `o1` dans la pile. Cet objet disparaîtra dès que se fermera l'accolade dont l'ouverture précède juste sa création.

De même, si vous passez un objet comme argument, automatiquement un nouvel objet sera créé, copie de celui que vous désirez passer. Une différence clé avec Java, Python et PHP 5 est qu'étant donné qu'il s'agit de la copie du référent et non pas de l'objet, la méthode, dans ces trois langages, agira bien sur l'objet original et non pas sur une copie toute fraîche, mais destinée à disparaître une fois la méthode terminée, comme en C++ et C#. Comme illustré par les codes qui suivent, il est donc possible en C# et C++ de bénéficier du même mode de gestion de mémoire pile des variables non-objets, et ce pour les objets.

En C++

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
public:
  O1() /* constructeur */ {
    cout << "un nouvel objet O1 est cree" << endl;
  }
  O1(const O1 &uneCopieO1) /* constructeur par copie */{
    cout << "un nouvel objet O1 est cree par copie" << endl;
  }
  ~O1() /* destructeur */{
    cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
  }
  void jeTravaillePourO1() {}
};
void usageO1(O1 unO1){
  unO1.jeTravaillePourO1();
}
int main(int argc, char* argv[]){
  O1 unO1; /* je crée un objet O1 */
  usageO1(unO1); /* la méthode reçoit une copie de cet objet */
  return 0;
}
```

Résultat

```
un nouvel objet O1 est créé
un nouvel objet O1 est créé par copie
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O1 se meurt...
```

Il faut, pour comprendre ce code, découvrir l'existence de deux nouvelles méthodes particulières, appelées le `constructeur par copie` et le `destructeur`. La première est appelée, automatiquement, dès qu'un objet se trouve dupliqué, notamment lors du passage d'argument. Elle permet, comme nous le comprendrons mieux dans les chapitres 10 et 14, de transformer une copie de surface en une copie profonde. Ici, ce constructeur se borne à signaler qu'on fait appel à lui.

Le destructeur, quant à lui, est une méthode appelée, automatiquement, dès la destruction d'un objet. Cette méthode ne peut recevoir d'argument car le programmeur n'est pas à l'origine de son appel. Là aussi, nous

comprendrons mieux l'importance de son rôle par la suite et dès le prochain chapitre. Elle est appelée juste avant la destruction de l'objet et permet de libérer certaines ressources référées par celui-ci avant de le faire disparaître. Ici, de même, ce destructeur se borne à signaler son appel. On voit que deux objets sont créés et détruits dans l'exécution de ce code, sans qu'il soit nécessaire de les détruire par une instruction explicite. Le second est créé lors du passage comme argument du premier. Il en est une copie. Toute cette mémoire est gérée par un système de pile, et les objets disparaissent dès la fermeture des accolades, le premier à la fin de la procédure `usage01()`, le second à la fin du programme.

En C#

```
using System;
public struct O1 /* ATTENTION ! On utilise une structure plutôt qu'une classe */{
  private int a;

  public void jeTravaillePourO1() {
    a = 5;  // modifie l'attribut
  }
  public void donneA(){
    Console.WriteLine("la valeur de a est: " + a);
  }
}
public class TestMemoirePile {
  public static void Test(O1 unO1){
    O1 unAutreO1 = new O1();
    unAutreO1.jeTravaillePourO1();
    unO1.jeTravaillePourO1();
  } // la copie de unO1 et l'objet unAutreO1 disparaissent ici.
  public static void Main(){
    O1 unO1 = new O1();
    unO1.donneA();
    Test(unO1);
    unO1.donneA(); /* On retrouve la valeur de l'attribut a de
        l'objet de départ, malgré le passage comme argument dans la méthode Test() */
  }
}
```

Résultat

```
la valeur de a est : 0
la valeur de a est : 0
```

Dans le code C# qui précède, nous utilisons une structure en lieu et place de classe. Cela nous permet de traiter les objets issus de ces structures exactement comme n'importe quelle variable de type prédéfini. Notez qu'aucun destructeur ne peut être déclaré dans une structure d'où son absence dans notre code ici.

Ainsi, dans le code, trois objets instance de la structure `O1` sont créés. L'un des trois est créé lors du passage par argument, et on constate que la modification de son seul attribut `a` n'affecte pas l'objet original dont il n'est qu'une simple copie. Tant la copie passée par argument que les deux autres objets créés comme variable locale de la méthode `Test(O1)` et de la méthode `Main()` disparaîtront également dès la fermeture des accolades.

Structure en C#

En C#, les objets issus d’une structure sont traités directement par valeur, dans la mémoire pile, et sans référent intermédiaire. Ils le sont comme n’importe quelle variable de type prédéfini. Les structures sont utilisées en priorité pour des objets que l’on veut et que l’on sait temporaires. Un constructeur par défaut y est prévu et donc on ne peut le surcharger en en définissant explicitement un autre. Plus important encore, les structures ne peuvent hériter entre elles, bien qu’elles héritent toutes de la classe Objet. En revanche, elles peuvent implémenter des interfaces. Tout cela s’explique aisément lorsqu’on sait que les structures sont exploitées par valeur et non par référent, et que les mécanismes d’héritage et de polymorphisme sont plus faciles à réaliser pour des objets uniquement adressés par leur référents. Il est plus facile de s’échanger les référents que les objets dans leur entièreté.

C# et Anders Hejlsberg

Se retrouver au côté de Bill Gates en février 2002 à San Francisco, pour annoncer la mise sur le marché d’un nouveau logiciel Microsoft, Visual Studio .Net, présenté comme révolutionnaire car colonne vertébrale de toute une stratégie à venir et dénommée .Net, n’est pas donné au premier quidam venu. Anders Hejlsberg, de fait, n’en est pas un. Concepteur principal du langage C#, innovation capitale dans l’environnement de programmation Microsoft, Hejlsberg n’en est pas à son premier coup de maître. Danois d’origine, pays décidément fournisseur de grandes sirènes de la programmation, avant de rejoindre Microsoft en 1996, il passe quelques années chez Borland comme créateur du Turbo Pascal et en tant que leader de l’équipe à l’origine de Delphi (un autre langage OO bien connu). Chez Microsoft, il prend une part active au développement du Visual J++, habillage Microsoft de Java. On connaît les déboires que connu ce langage, hybride très habile de la syntaxe de Java avec les librairies de Microsoft, et qui irrita Sun au point de mettre la justice sur le coup. Mieux valait pour Microsoft renommer la main basse effectuée sur Java, afin de donner le jour à C# (prononcé C Sharp). Le « J » s’est, comme par magie, transformé en « C ».

Évidemment, le nom prête à plaisanteries et elles ne manquent pas. Ce langage s’est déjà vu traité de Visual J- - ou de C bémol. C’est sans doute sa ressemblance avec Java qui le rend le plus vulnérable à ces agressions. Et nous savons la profonde et indéfectible amitié qui lie Bill Gates à Scott McNeal, dirigeant de Sun. Dans la bouche de ce dernier .Net devient .Not, .Not yet ou .Nut. Rien d’étonnant à qui déclare aussi que « dans cette guerre sans merci, nous récupèrerons chaque développeur et ne le laisserons pas s’abandonner du côté sombre ». Plusieurs fois dans notre ouvrage, nous constatons, parfois avec agacement, ces similitudes dont se défend pourtant notre auteur (stratégie et marketing obligent). En effet, dans les premiers écrits consacrés à ce nouveau langage, il était très difficile de trouver une simple mention à Java. Le langage était perçu comme une évolution naturelle de C++, mais l’absence d’allusion à Java ramenait celui-ci à un statut de véritable « chaînon manquant ».

S’il est vrai que C# s’est moins éloigné de C++ que Java ne l’a fait (on y retrouve davantage de types de données communs à C++, des objets stockables en mémoire pile, une prédominance du typage statique, et on peut même y intégrer, à ses risques et périls, des pointeurs C++), il n’en reste pas moins vrai que son plus proche voisin demeure Java et non pas C++. Et pour cause, le langage récupère la cohérence OO héritée de Smalltalk, mais, tout comme Java, base cet habillage OO sur une syntaxe C. On y retrouve le ramasse-miettes et une interprétation du code plutôt qu’une compilation, interprétation qui se transforme, cependant, en une véritable compilation dès la première exécution du code (les performances sont alors améliorées). Ce niveau intermédiaire permet une communication facilitée entre différents langages de programmation.

Est-il meilleur que Java ? Voilà le type même de question à laquelle il est impossible de répondre, et cela l’est également pour tous les langages que nous traitons dans ce livre. L’Italie est-elle meilleure que la France ? La paëlla est-elle meilleure que le couscous ? C# a pour lui d’apparaître cinq ans après Java et de pouvoir puiser çà et là ce qui se fait de mieux dans les étals de plusieurs langages. Sans doute a-t-il passé un peu plus de temps au rayon Sun et, postérieur à Java, il a pu éviter certaines maladresses de celui-ci dont se plaignent les programmeurs et que Java ne peut supprimer par souci de compatibilité avec l’existant. C# intègre, par exemple, des aspects de VB, comme les mécanismes d’accès aux attributs. Son jeune âge lui permet aussi d’incorporer des éléments technologiques plus modernes. Il en va ainsi de la totale prise en compte d’XML, langage universel de description de contenu de documents publiés sur le Web. Il est possible, à partir du code, de générer très facilement une description de son contenu en XML. Hejlsberg le décrit comme le premier langage facilitant véritablement la programmation à base de composants, bien qu’il reste généralement très évasif sur ce qu’il entend par là, et en quoi ni Java ni d’autres ne pourraient servir à la programmation de ces mêmes composants.

Il est clair que, quitte à s'embarquer sur la nouvelle plate-forme Microsoft de développement Internet, le langage C# apparaît comme un outil de développement de choix (il est d'ailleurs recommandé par Microsoft). Nous verrons dans le chapitre 16 sa prise en compte très simple et très naturelle des services web, version XML des objets distribués communiquant à travers le Web. De fait, Hejlsberg présente toujours son langage comme partie intégrante de .Net, en le plébiscitant comme un des éléments clés de cette énorme boîte à outils de développement Internet. .Net permet, en effet, un développement facilité et transparent d'applications distribuées, par l'entremise des services Web générés automatiquement à partir des codes sources. .Net, dont la vocation première est de faciliter la conception d'applications distriuées sur le Web par l'entremise d'ASP.Net, lorsque ces applications se parlent via un browser, ou par service web quand les objets communiquent directement par envoi de message d'une application à l'autre, a enrichi la description sémantique de ces interactions par l'utilisation abondante et largement automatisée du langage XML. Cette intégration a permis à Microsoft de prendre quelques longueurs d'avance par rapport à son concurrent direct, Sun.

Malgré son « interprétabilité » (commune à Java), C# pour l'instant ne tourne que sur Windows. Microsoft parle d'universalité de cette plate-forme mais dans un sens nouveau. La version interprétable du langage (CLR – Common Langage Runtime) est partageable avec de nombreux autres langages supportés par .Net (vingt-deux à ce jour), comme C++, VB .Net, Jscript, Cobol, Eiffel (d'où la participation de Meyer), Perl, Python, Smalltalk et d'autres à venir. Cela permet à un code C# d'hériter éventuellement d'une classe préalablement développée en VB .Net, et d'envoyer un message à une classe développée en Eiffel. Cela est possible, car tous ces langages se conforment à une CLS (Common Langage Specification – d'où, de fait, la nouvelle version de VB) qui permet de passer de l'un à l'autre. Là où Java est mono-langue mais multi- plates-formes, .Net est multi-langues mais mono-plate-forme. Une tentative actuelle de standardisation de C# est en cours. Parlons aussi du projet plus récent Mono destiné à pourvoir une version Open Source de .Net qui sera susceptible de tourner sur des plates-formes aussi variées que Linux, Solaris, Mac OS X.

La version 2 de .NET, langage et environnement, s'était illustrée par la prise en charge de la généricité traitée au chapitre 21. La version 3 de .NET connaît une autre avancée majeure, la bibliothèque Linq (décrite au chapitre 19) qui propose une nouvelle méthode – une de plus – pour améliorer la correspondance entre le monde des bases de données relationnelles et l'orienté objet. Linq propose un langage d'interrogation, inspiré du SQL, capable de s'interfacer indifférement avec des collections d'objets, ces mêmes objets étant stockés soit sous forme XML soit dans une base de donnée relationelle.

Visual C++.Net

C++ étant largement étudié dans cet ouvrage, nous nous en voudrions de ne pas dire un petit mot sur Visual C++.Net, la nouvelle mouture de ce langage, une version assez radicalement remaniée grâce à son intégration à .Net et à cause de la nécessité de se rendre compatible aux vingt-et-un autres langages supportés par la plate-forme. Ce nouveau langage est étonnament puissant car, par exemple, il combine les systèmes de gestion mémoire par ramasse-miettes (on parle alors de `_gc_class` plutôt que de `class`, et toute la sophistication consiste à mêler ces deux types de class et la gestion mémoire différente qui s'y rapporte) et par instruction directe du programmeur. De même que C++ ne présente pas la même offre en librairies que Java (Multithread, GUI, bases de données), ce nouvel arrivant intègre idéalement toutes les librairies .Net, ce qui le rend aussi riche en services et librairies que Java. (Ces librairies sont de fait communes à tous les langages de la plate-forme. Vous pouvez en voir quelques-unes – Ado.Net, les services web ou les GUI – à l'œuvre dans certains chapitres de ce livre.)

Il est difficile, vu son jeune âge, de vous donner une liste définitive de manuels de programmation C#. Comme introduction très rapide et très économique au langage, on peut citer :

- *Le Langage C#*, Christine EBEHARDT, Campus Press
- *Introduction à C#*, Pierre-Yves SAUMONT et Antoine MIRECOURT, Osman Eyrolles MultiMedia.
- Une description rapide mais approfondie (parfaite pour les programmeurs Java) se trouve dans : *C# Essentials*, Ben ALBARHI, Peter DRAYTON et Brad MERRIL, O'Reilly.
- DEITTEL et DEITELL ne sont évidemment pas en reste et ont récemment publié *C# how to program* dans la collection Prentice Hall ainsi qu'une version plus avancée ; C# for Experienced Programmers.
- Le très bon *Programming C#* de Jesse LIBERTY chez O'Reilly et *C# And Object Orientation* de John HUNT chez Springer.

Et enfin pour C# dans le contexte .Net :
– *Microsoft .Net for Programmers*, Fergal GRIMES, Manning
– *Beginning Asp.Net using C#*, Chris ULLMAN, Chris GOODE, Juan T. LLIBRE et Ollie CORNES, Wrox.
– *Microsoft. NET for Programmers*, Fergal GRIMES, Manning.

Disparaître de la mémoire comme de la vie réelle

Ce mode de gestion de la mémoire pile est intimement lié à une organisation procédurale de la programmation, où le programme est décomposé en procédures ou en blocs imbriqués, lesquels nécessiteront, uniquement pendant leur déroulement, un ensemble de variables, qui seront éliminées à la fin. Nous avons vu que la programmation OO se détache de cette vision, en privilégiant les objets aux fonctions. Il est, en conséquence, tout aussi important de détacher le temps de vie des objets de leur participation à certaines fonctions précises. L'esprit de l'OO est qu'un objet devient encombrant si, dans le scénario même que reproduit le programme, l'objet réel, que son modèle informatique « interprète », disparaît tout autant de la réalité. Dans le petit écosystème vu précédemment, la proie disparaît quand elle se fait manger par le prédateur, l'eau disparaît quand sa quantité devient nulle.

Représentez-vous tous ces jeux informatiques, dans lesquels des balles apparaissent et disparaissent, des avions explosent, des héros meurent, des footballeurs quittent le terrain. À chaque fois, l'objet représenté disparaît, tant et si bien que son élimination de la mémoire est même souhaitée, pour permettre à un nouvel objet d'exister et prendre sa place. Il est bien plus difficile d'organiser cette gestion de la mémoire par un mécanisme de pile car, une fois l'objet créé, son temps de vie peut transcender plusieurs blocs fonctionnels, pour ne finalement disparaître, éventuellement de manière conditionnelle, dans l'un d'entre eux (et pas du tout automatiquement dans le bloc où il fut créé). L'OO permet aux objets de vivre bien plus longtemps (et ils vous en remercient) et, surtout, rend leur élimination indépendante des fonctions qui les manipulent, mais plus dépendante du scénario qui se déroule, aussi inattendu soit-il.

La vie des objets indépendante de ce qu'ils font

L'orienté objet, se détachant d'une vision procédurale de la programmation, tend à rendre indépendante la gestion mémoire occupée par les objets de leur participation dans l'une ou l'autre opération. Cette nouvelle gestion mémoire résultera d'un suivi des différentes transformations subies par l'objet et sera, soit laissée à la responsabilité du programmeur, soit automatisée.

Mémoire tas

Tous les langages OO permettent donc un mode de création et de destruction d'objets autrement plus flexible que la mémoire pile. En C++ et C#, ce nouveau mode est en complément de la mémoire pile. En Java, PHP 5 et Python, ce nouveau mode est le seul possible pour les objets, la mémoire pile restant, en revanche, la seule possibilité pour toutes les autres variables de type prédéfini ou primitif. Dans ce mode plus flexible et en C++, C, PHP 5 et Java, tous marqués à vie par leur précurseur, le C, on peut créer les objets n'importe où dans le programme par le truchement du `new` (en Python, `new` n'est plus nécessaire). Ils seront créés n'importe quand et pourront être installés partout où cela est possible dans la mémoire, d'où la nécessité d'un référent qui connaisse leur adresse et permette de les retrouver et les utiliser. Mais comment fera-t-on disparaître un objet ? Simplement, quand la « petite histoire » que raconte le programme l'exige ? De nouveau, il est nécessaire de différencier deux politiques : celle très libérale du C++, qui laisse au seul programmeur le soin de décider de la vie et de la mort des objets, et le mode étatisé des quatre autres langages, qui s'en occupe pour vous, en arrière-plan.

C++ : le programmeur est le seul maître à bord

En C++, vous pouvez, n'importe où dans un programme, supprimer un objet qui a été créé par `new`, en appliquant sur son référent l'instruction `delete`. Vous devenez les seuls maîtres à bord, et, à ce titre, capable du meilleur comme du pire. Pour le meilleur, on vous fait confiance, malheureusement, pour le pire, on vous conserve cette confiance. Ainsi, voici deux scénarios catastrophes, toute proportion gardée bien entendu, que les programmeurs C++ reconnaîtront aisément, même s'ils s'en défendent.

Un premier petit scénario catastrophe en C++

```
#include "stdafx.h"
#include "iostream.h"
class O1{
private:
  int a;
public:
  O1() /* constructeur */{
    a = 5;
    cout << "un nouvel objet O1 est cree" << endl;
  }
  O1(const O1 &uneCopieO1) /* constructeur par copie */{
      cout << "un nouvel objet O1 est cree par copie" << endl;
  }
  ~O1() /* destructeur */{
    cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
  }
  void jeTravaillePourO1() {
    cout << "a vaut: "<< a << endl;
  }
};
void jeTueObjet(O1 *unO1){
  delete unO1; // on efface l'objet O1
}
void jeCreeObjet(){
  O1 *unO1 = new O1();
  jeTueObjet(unO1);
  unO1->jeTravaillePourO1(); //l'objet a disparu bien que son utilisation reste parfaitement
  ➥possible.
}
int main(int argc, char* argv[]){
  jeCreeObjet();
  return 0;
}
```

Résultats

```
un nouvel objet O1 est créé
aaahhhh... un objet O1 se meurt...
a vaut : - 572662307
```

Un même objet unO1 est référencé deux fois. Dans la procédure jeCreeObjet() (on parlera de procédure ou de fonction car elle est définie en dehors de toute classe), on crée d’abord l’objet unO1, et puis on le passe en argument de la méthode jeTueObjet(), qui s’empresse de l’effacer. Mais alors qu’il est éliminé par la méthode jeTueObjet(), il est encore référencé dans la méthode jeCreeObjet() par le référent unO1. Comme l’objet référé par ce référent a disparu, ce dernier se mettra à référer n’importe quoi dans la mémoire, avec toutes les mauvaises surprises dont les programmeurs du C++ sont friands.

Vous voyez, par exemple, qu’au lieu d’afficher la valeur 5, ce à quoi on aimerait s’attendre, c’est une valeur complètement imprévue qui apparaît. Rien dans la compilation du programme n’a pu prévenir ce dysfonctionnement. Évidemment, dans ce petit code, l’endroit où est créé l’objet est tellement proche de l’endroit où celui-ci est détruit qu’une telle situation vous semble parfaitement improbable. Détrompez-vous ! Dans un code plus grand et bien plus réparti entre les programmeurs, un de ces derniers, dans l’écriture de sa méthode, aura tout loisir de détruire un objet encore utile à un tas d’autres programmeurs. Nous avons vu que l’adressage indirect, permettant à un même objet d’être référé de multiples fois (dans des contextes différents), est un mécanisme inhérent au paradigme objet. Chaque référent devient alors disponible pour effacer l’objet, quand bien même les autres référents, tout perdus qu’ils soient, persisteraient à y faire référence ! Les référents fous ou pointeurs fous sont légion en C++, et aucune voiture d’ambulanciers ne se charge de les récupérer dans la mémoire. Un autre scénario tout aussi dramatique est celui qui consiste à effacer plusieurs fois un même objet par la répétition de l’instruction delete sur un même référent. Les référents fous ou pointeurs fous sont légion en C++, et aucune voiture d’ambulanciers ne se charge de les récupérer dans la mémoire. Un autre scénario tout aussi dramatique est celui qui consiste à effacer plusieurs fois un même objet par la répétition de l’instruction delete sur un même référent.

La mémoire a des fuites

Rappelez-vous le petit laïus moralisateur au début du chapitre, vous incitant à ne pas gaspiller la mémoire des ordinateurs, sauf à la jeter dans un sac poubelle prévu à cet effet. Il est très fréquent, dans les langages où vous êtes responsables de la gestion mémoire, de laisser traîner des objets devenus inaccessibles, donc parfaitement encombrants. On parle alors de « fuite de mémoire ». Ce scénario porte moins à conséquence que le précédent. C’est même le scénario parfaitement inverse car, maintenant, alors que les objets existent encore, ils sont devenus hors de portée. Ils ralentissent le code et font chuter les performances, mais n’occasionnent rien de totalement imprévisible.

Second petit scénario catastrophe en C++

```
#include "stdafx.h"
#include "iostream.h"
class O2{
public:
   O2(){
     cout << "un nouvel objet O2 est cree" << endl;
   }
  ~O2() /* destructeur */{
    cout <<"aaahhhh ... un objet O2 se meurt ..." << endl;
   }
};
class O1{
private:
  O2 *monO2; /* on agrège un objet O2 dans O1 */
public:
   O1() /* constructeur */{
```

```
      cout << "un nouvel objet O1 est cree" << endl;
      monO2 = new O2(); /* on crée ici l'objet O2 */
    }
    ~O1() /* destructeur */{
     cout <<"aaahhhh ... un objet O1 se meurt ..." << endl;
    }
};
int main(int argc, char* argv[]){
  O1 *unO1 = new O1();
  unO1 = new O1(); /* on ré-utilise le même référent */
  delete unO1;
  return 0;
}
```

Résultats

```
un nouvel objet O1 est créé
un nouvel objet O2 est créé
un nouvel objet O1 est créé
un nouvel objet O2 est créé
aaahhhh... un objet O1 se meurt...
```

La figure 9-2 montre ce qui se passe dans la mémoire des objets lorsque le programme procède à la destruction du dernier objet `O1` référé. Trois objets continueront à encombrer la mémoire inutilement, mémoire gaspillée et irrécupérable. La première raison en est l'utilisation du même référent `unO1` pour les deux objets créés. Alors que deux objets `O1` sont créés, seul le second sera accessible, car le même référent que celui utilisé pour le premier objet est exploité à nouveau. De ce fait, comme l'attribut `monO2 du` premier objet `O1` pointe vers un second objet, les deux objets occuperont inutilement la mémoire. Lorsque grâce au `delete`, vous effacez le second `O1`, en fait vous n'effacez que cet objet et le référent vers son objet `O2`. Mais si vous omettez d'effacer

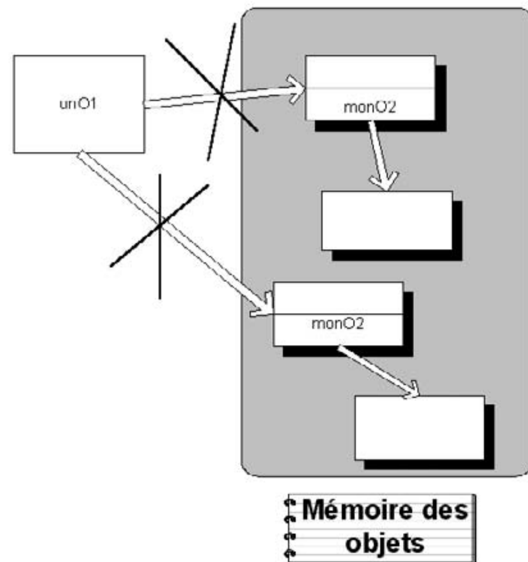


Figure 9-2

Le déroulement en mémoire des codes C++ et Java commentés dans le texte. Le référent « unO1 » pointe d'abord sur un premier objet O1 (chaque objet O1 pointe à son tour sur un objet O2) pour ensuite se mettre à pointer sur un autre objet O1 avant de passer à null. En C++, seul le troisième objet dans la mémoire sera effacé. En Java, C# et Python grâce au comptage des référents et au ramasse-miettes, tous les objets finiront par être effacés de la mémoire.

l'objet O2 également (ce que vous pouvez faire par un mécanisme qui sera détaillé dans le prochain chapitre, et qui consiste à redéfinir le destructeur), celui-ci, à son tour, occupera inutilement la mémoire.

En substance, un langage comme C++, qui vous espère adulte et baptisé en matière de gestion de mémoire, a tendance à quelque peu surestimer ses programmeurs. Et ceux-ci se retrouvent, soit avec des référents fous, qui se mettent à référer de manière imprévisible tout et n'importe quoi dans la mémoire, soit avec des objets perdus, comme des satellites égarés dans l'espace, sans aucun espoir de récupération.

En Java, C#, Python et PHP 5 : la chasse au gaspi

D'autres langages, comme Java, C#, Python et PHP 5, se montrent moins confiants quant à vos talents de programmeurs, et préfèrent prévenir que guérir. Ils partent de l'idée toute simple qu'un objet n'est plus utile dès lors qu'il ne peut plus être référencé. Un objet devenu inaccessible ne demande qu'à vous restituer la place qu'il occupait. L'idéal serait donc de réussir à vous débarrasser, à votre insu (mais tout à votre bénéfice), pendant l'exécution du programme, des objets devenus encombrants. Une manière simple est de rajouter, comme attribut caché de chaque objet, un compteur de référents, et de supprimer tout objet dès que ce compteur devient nul. En effet, un objet débarrassé de tout référent est inaccessible, donc inutilisable, et donc bon à jeter.

Si vous vous repenchez sur les deux petits codes présentés précédemment, vous verrez que ce seul mécanisme vous aurait évité, d'abord, le référent fou (puisque vous ne pouvez vous-même effacer un objet, un référent fou devient impossible), ensuite la perte de mémoire. En effet, le premier objet O1 disparaît car il n'est plus référencé par le référent unO1. Avec lui, disparaît également le référent vers l'objet O2, entraînant donc ce dernier dans sa perte. Par exemple, le petit code Java suivant, équivalent, dans l'esprit, au code C++ précédent, vous montre que tous les objets inaccessibles seront en effet détruits. La méthode finalize() joue, en substance, le même rôle que le destructeur en C++, et s'exécute lors de la destruction de l'objet. Nous reviendrons sur son rôle dans les prochains chapitres.

En Java

```
class O2{
  public O2() /* constructeur */{
    System.out.println("un nouvel objet O2 est cree");
  }
  protected void finalize () /* le destructeur */{
    System.out.println("aaahhhh ... un objet O2 se meurt ...");
  }
}
class O1{
  private O2 monO2; /* on agrège un objet O2 dans O1 */

  public O1() { /* constructeur */
    System.out.println("un nouvel objet O1 est cree");
    monO2 = new O2(); /* on crée ici l'objet O2 */
  }
  protected void finalize() { /* destructeur */
    System.out.println("aaahhhh ... un objet O1 se meurt ...");
  }
}
```

```
public class TestFuiteMemoire{
  public static void main(String[] args){
    O1 unO1 = new O1();
    unO1 = new O1(); /* on ré-utilise le même référent */
    unO1 = null;
    System.gc(); /* appel explicite du garbage-collector */
  }
}
```

Résultats

```
un nouvel objet O1 est créé
un nouvel objet O2 est créé
un nouvel objet O1 est créé
un nouvel objet O2 est créé
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
```

En C#

```
using System;
class O2{
  public O2() /* constructeur */{
    Console.WriteLine("un nouvel objet O2 est cree");
  }
    ~ O2() /* le destructeur */{
    Console.WriteLine("aaahhhh ... un objet O2 se meurt ...");
  }
}
class O1{
  private O2 monO2; /* on agrège un objet O2 dans O1 */
  public O1() { /* constructeur */
    Console.WriteLine("un nouvel objet O1 est cree");
    monO2 = new O2(); /* on crée ici l'objet O2 */
  }
    ~ O1() { /* destructeur */
    Console.WriteLine("aaahhhh ... un objet O1 se meurt ...");
  }
}
public class TestFuiteMemoire{
  public static void Main(){
    O1 unO1 = new O1();
    unO1 = new O1(); /* on réutilise le même référent */
    unO1 = null;
    GC.Collect(); /* appel explicite du garbage-collector */
  }
}
```

Le code C#, parfaitement équivalent au code Java, est présenté de manière à indiquer les quelques différences d'écriture avec Java, notamment dans la syntaxe du destructeur, qui rappelle plutôt celle du C++.

En PHP 5

Rien de bien original dans la version PHP 5 du même code qui produira, là encore, le même résultat.

```
<html>
<head>
<title> Gestion mémoire des objets </title>
</head>
<body>
<h1> Gestion mémoire des objets </h1>
<br>

<?php
   class O2 {

           public function __construct() {
                       print ("un nouvel objet O2 est cree <br> \n");
           }

           public function __destruct () { // déclaration du destructeur
                       print ("aaahhhh .... un objet O2 se meurt ... <br> \n");
           }

   }

   class O1 {
           private $monO2;

           public function __construct() {
                       print ("un nouvel objet O1 est cree <br>\n");
                       $this->monO2 = new O2();


   }
           public function __destruct () {
                       print ("aaahhhh .... un objet O1 se meurt ... <br> \n");
           }
   }

   $unO1 = new O1();
   $unO1 = new O1();
   $unO1 = NULL;

   ?>

</body>
</html>
```

Le ramasse-miettes (ou garbage collector)

On peut voir qu'au contraire du C++, dans les trois autres langages, tous les objets encombrants sont effacés :

- le premier objet `O1`, car on réutilise son référent pour la création d'un autre objet ;
- le second car on a mis son référent à `null`.

Cette dernière instruction est utile lorsque l'on cherche à se débarrasser d'un objet devenu encombrant : il suffit d'assigner la valeur `null` à son référent. À la différence de C++, les autres langages n'autorisent pas une suppression d'objet par une simple instruction. La dernière instruction du programme ne se rencontre en général pas, car il y est explicitement fait appel à l'effaceur d'objets : le garbage collector. En général, cet appel se fait à une fréquence soutenue et calibrée par défaut par la machine virtuelle de Java, C#, Python ou PHP 5, dès que la mémoire RAM commence à être sérieusement occupée. Ce calibrage suffit dans la plupart des cas, mais vous avez néanmoins la possibilité d'interférer directement avec ce mécanisme, comme indiqué dans les codes.

Le mécanisme responsable de la découverte et de l'élimination des objets perdus s'appelle, en effet, le garbage collector, traduisible par « camion-poubelle » ou « ramasse-miettes ». Le ramasse-miettes passe en revue tous les objets de la mémoire avec pour mission d'effacer ceux qui possèdent un compteur de référents nul. En général, il s'exécute en parallèle (c'est-à-dire sur un thread à part) avec votre programme. (Nous découvrirons le multithreading dans le chapitre 17.)

Souvent, celui-ci se déclenchera naturellement, lorsqu'on commence à remplir la mémoire de manière conséquente. Il est quelquefois paramétrable, selon que vous le souhaitiez hyperactif et donc très concentré sur les économies à réaliser dans la mémoire, ou plus laxiste. Il est clair qu'un compromis subtil est à rechercher ici, car souvent les économies mémoire permettent une accélération du programme. Mais si le prix à payer pour récupérer cette précieuse mémoire est un ralentissement encore plus important du programme, occasionné par le fonctionnement simultané du ramasse-miettes et de votre programme, vous en percevez aisément le ridicule (qui ne tue ni les êtres humains ni les objets malheureusement).

Des objets qui se mordent la queue

Un seul problème subsiste et, hélas, non des moindres : ce compteur de référents peut être non nul pour certains objets, bien que ces objets en question soient non accessibles. Il s'agit de tous les objets impliqués dans des structures relationnelles présentant des cycles, comme dans la figure 9-3.

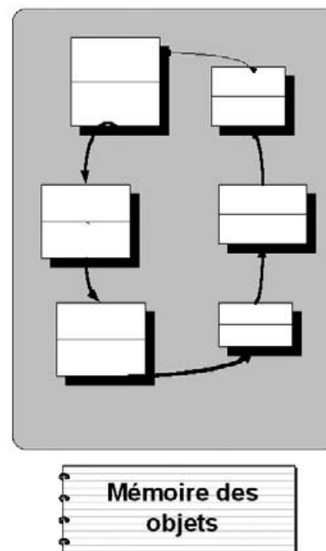


Figure 9-3

Tous les objets sont référés au moins une fois, mais le cycle entier d'objets est inaccessible.

Cette situation est plus que fréquente en OO, car il suffit par exemple que deux objets, comme notre proie et prédateur, se réfèrent mutuellement pour que cela se produise. La détection de ces cycles nécessite une très laborieuse exploration de la mémoire, qui a finalement l’effet pervers, si elle se déroule simultanément à l’exécution du programme, de ralentir celui-ci. Les langages qui ont opté pour la manière automatique de récupération de mémoire ont donc inventé des systèmes ingénieux, dont la description dépasserait le cadre de cet ouvrage, pour parer au mieux à ce problème.

Ils sont sûrs de ce qu’ils font, en ceci qu’aucun objet utile ne peut disparaître, et qu’aucun référent ne puisse être atteint de soudaine folie. Cependant, ils acceptent de ne pas être parfaits et exhaustifs, en abandonnant dans la mémoire quelques objets qui sont devenus inutiles. Par exemple, ils choisissent de ne pas systématiquement passer toute la mémoire en revue, à la recherche des objets perdus, mais uniquement de se concentrer sur les objets les plus récemment créés. Les objets dont la création récente résulte d’une utilisation temporaire pour une fonctionnalité précise seront d’excellents candidats à une élimination rapide.

Une dernière opération à réaliser, une fois la mémoire récupérée, est de ré-organiser celle-ci de façon à très facilement repérer les zones occupées et inoccupées. Cette opération aura pour effet de déplacer les objets du programme afin de les compacter dans la mémoire. Ce recompactage des objets permet d’exploiter au mieux les systèmes de mémoire hiérarchique, tels que la mémoire cache. En effet, la chance que les objets agissant de concert se trouvent localisés dans une zone voisine s’accroît, en les installant les uns à côté des autres. Il faudra, à votre insu (mais c’est autant de gagné bien sûr), changer les adresses contenues dans les référents. Ce ramasse-miettes existant en Java, C#, Python, PHP 5 et originellement dans LISP, est donc un procédé d’une grande sophistication, tentant au mieux de vous éviter les terribles méprises ou gaspillages inhérents au C++, tout en « prenant conscience » de son coût en temps calcul, et des moyens de diminuer celui-ci. La claque, quoi !

Nous finissons le chapitre en présentant la même version des codes Java, C# et PHP 5 mais cette fois-ci en Python, afin d’expliquer davantage le rôle du mot-clé `self`. Les résultats des deux versions du code sont indiqués en dessous de celui-ci.

En Python

```
import gc # imbrication dans le code des fonctionnalités du ramasse-miettes
class O2:
    def __init__(self):
        print "un nouvel objet O2 est cree"
    def __del__(self):
        print "aaahhhh ... un objet O2 se meurt ..."
class O1:
    __monO2 = None
    def __init__(self):
        print "un nouvel objet O1 est cree"
        # self.__monO2 = O2() # première version du code
        # __monO2 = O2() # deuxième version du code
    def __del__(self):
        print "aaahhhh ... un objet O1 se meurt ..."
unO1 = O1()
unO1 = O1()
unO1 = None
gc.collect()
```

Résultat de la première version du code, avec le self, comme dans les exemples Java et C#

```
un nouvel objet 01 est cree
un nouvel objet 02 est cree
un nouvel objet 01 est cree
un nouvel objet 02 est cree
aaahhhh ... un objet 01 se meurt ...
aaahhhh ... un objet 02 se meurt ...
aaahhhh ... un objet 01 se meurt ...
aaahhhh ... un objet 02 se meurt ...
```

Résultat de la deuxième version du code, sans le self

```
un nouvel objet 01 est cree
un nouvel objet 02 est cree
aaahhhh ... un objet 02 se meurt ...
un nouvel objet 01 est cree
un nouvel objet 02 est cree
aaahhhh ... un objet 02 se meurt ...
aaahhhh ... un objet 01 se meurt ...
aaahhhh ... un objet 01 se meurt ...
```

La première version du code est identique à celle des codes Java et C#. Cependant, dès que l'on supprime le `self`, le référent `__mon02` ne réfère plus l'attribut de la classe, mais un nouvel objet, variable locale du constructeur et qui se borne à disparaître dès que le constructeur a fini de s'exécuter. C'est la présence du `self` (en tout point semblable à la présence du `$this->` en PHP 5) qui permet de différencier l'appel explicite aux attributs de l'objet de la création et l'utilisation de variables locales aux méthodes. Le paramètre self est donc indispensable, comme dans la plupart des codes Python qui précèdent, dès que l'on utilise les attributs propres à l'objet. L'omettre entraîne la création et la manipulation de variables locales aux méthodes.

Afin de clarifier davantage encore la façon subtile dont Python différencie, dans ses classes variables locales, attributs de classe et attributs d'objet, le petit code suivant devrait vous être très utile.

En Python

```
class 01:
    a=0
    b=0
    c=0
    def test(self):
        a=1 #cela reste une variable locale car elle n'est liée à rien lors de son affectation
        01.b=2 #cela reste un attribut de classe car il est référé comme tel
        self.c=3 #cela devient, grâce à la présence de self, un attribut d'objet
        print a,01.b,01.c
un01 = 01()
un01.test()
print 01.a,un01.a
print 01.b,un01.b
print 01.c,un01.c
```

Résultats

```
1 2 0
0 0
2 2
0 3 #ici, on fait bien la différence entre « c » attribut de classe et « c » attribut d'objet
```

Le garbage collector ou ramasse-miettes

Il s'agit de ce mécanisme puissant, existant dans Java, C#, Python et PHP 5, qui permet de libérer le programmeur du souci de la suppression explicite des objets encombrants. Il s'effectue au prix d'une exploration continue de la mémoire, simultanée à l'exécution du programme, à la recherche des compteurs de référents nuls (un compteur de référents existe pour chaque objet) et des structures relationnelles cycliques. Une manière classique de l'accélérer est de limiter son exploration aux objets les plus récemment créés. Ce ramasse-miettes est manipulable de l'intérieur du programme et peut être, de fait, appelé ou simplement désactivé (dans les trois langages, ce sont des méthodes envoyées sur « gc » qui s'en occupent). Les partisans du C++ mettent en avant le coût énorme en temps de calcul et en performance occasionné par le fonctionnement du « ramasse-miettes ». Mais à nouveau, lorsqu'il est question de comparer le C++ aux autres langages (notez que des librairies existent qui permettent de rajouter un mécanime de garbage collector au C++), la chose s'avère délicate car les forces et les faiblesses ne portent en rien sur les mêmes aspects. C++ est un langage puissant et rapide, mais uniquement pour ceux qui ont choisi d'en maîtriser toute la puissance et la vitesse. Mettez une Ferrari dans les mains d'un conducteur qui n'a d'autres besoins et petits plaisirs que des sièges confortables, une voiture large et silencieuse ainsi qu'une complète sécurité, vous n'en ferez pas un homme heureux. Mettez un appareil photo aussi sophistiqué qu'un Hasselblad dans les mains d'un amateur qui n'a d'autres priorités que de faire rapidement et sur-le-champ des photos assez spontanées de ses vacances et les photos seront toutes ratées.

Exercices

Exercice 9.1

Expliquez pourquoi la mémoire RAM est une ressource précieuse, et pourquoi il est nécessaire de tenter au mieux de rassembler les objets dans cette mémoire.

Exercice 9.2

Dans le petit code C++ suivant, combien d'objets résideront-ils de façon inaccessible en mémoire, jusqu'à la fin du programme ?

```
#include "stdafx.h"
class O1 {
};
class O2 {
private:
  O1 *unO1;
public:
  O2()
  {
    unO1 = new O1();
  }
};
```

```
class O3 {
private:
  O2 *unO2;
public:
  O3(){
    unO2 = new O2();
  }
};
int main(int argc, char* argv[]) {
  O3 *unO3 = new O3();
  delete unO3;
  return 0;
}
```

Exercice 9.3

Dans un code équivalent en Java, tel que le code écrit ci-après, combien d'objets résideront encore en mémoire après l'appel au ramasse-miettes ?

```
class O1 {
}
class O2 {
  private O1 unO1;
  public O2(){
    unO1 = new O1();
  }
}
class O3 {
  private  O2 unO2;
  public O3(){
    unO2 = new O2();
  }
}
public class Principale {
  public static void main(String[] args) {
    O3 unO3 = new O3();
    unO3 = null;
    System.gc();
  }
}
```

Exercice 9.4

Indiquez ce que produirait le petit code C++ suivant, et expliquez pourquoi ce problème ne pourrait survenir en Java et en C# :

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
  int a;
```

```
public:
  O1() {
    a = 10;
  }
  void donneA() {
    cout << a << endl;
  }
};
class O2 {
public:
  O2(){
  }
  void jeTravaillePourO2(O1 *unO1) {
    delete unO1;
  }
  void jeTravailleAussiPourO2(O1 *unO1) {
    unO1->donneA();
  }
};
int main() {
  O1* unO1 = new O1();
  O2* unO2 = new O2();
  unO2->jeTravaillePourO2(unO1);
  unO2->jeTravailleAussiPourO2(unO1);
  return 0;
}
```

Exercice 9.5

Expliquez pourquoi le code Java suivant présente des difficultés pour le ramasse-miettes :

```
class O1 {
  private O3 unO3;
  public O1(){
    unO3 = new O3(this);
  }
}
class O2 {
  private O1 unO1;
  public O2(O1 unO1) {
    this.unO1 = unO1;
  }
}
class O3 {
  private  O2 unO2;

  public O3(O1 unO1) {
    unO2 = new O2(unO1);
  }
}
```

```
class O4 {
  private O1 unO1;
  public O4() {
    unO1 = new O1();
  }
}
public class Principale2 {
  public static void main(String[] args) {
    O4 unO4 = new O4();
    unO4 = null;
    System.gc();
  }
}
```

Exercice 9.6

Quel nombre affichera ce programme C++ à l'issue de son exécution ?

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
  static int morts;
public:
  static int getMorts(){
    return morts;
  }
public:
  ~O1(){
    morts++;
  }
};
int O1::morts = 0;

void essai(O1 unO1) {
  O1 unAutreO1;
}
int main(int argc, char* argv[]) {
  O1 unO1;
  for (int i=0; i<5; i++) {
    essai(unO1);
  }
  cout << O1::getMorts() << endl;
  return 0;
}
```

Exercice 9.7

Expliquez le fonctionnement du ramasse-miettes, les difficultés qu'il rencontre et les manières de l'accélérer.

Exercice 9.8

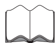
Expliquez comment et pourquoi dans la gestion mémoire des objets, C# tâche d'être un compromis parfait entre Java et C++.

10

UML 2

Ce chapitre est centré sur quelques diagrammes UML2, les plus importants pour la conception et le développement de programmes OO, tels le diagramme de classe et le diagramme de séquence. Par leur proximité avec le code (malgré leur expression graphique) et leur côté visuel (vu leur expression graphique), ils constituent un excellent support pédagogique à la pratique de l'OO. Ils sont devenus aujourd'hui incontournables pour la réalisation et la communication d'applications informatiques complexes.

Sommaire : Diagramme de classe — Diagramme de séquence — Les bienfaits d'UML



Doctus. — Il est temps d'aborder un aspect fondamental de la programmation objet. Elle ne constitue pas seulement une nouvelle manière d'organiser le travail de développement logiciel. Le découpage modulaire est en premier lieu le résultat d'un processus d'analyse et de conception…

Candidus. — … je te vois venir. Tu vas royalement m'annoncer que l'OO est l'architecture idéale pour réaliser un paradis virtuel et qu'un plan, décrivant les informations et les processus mis en jeu, peut être transformé en programme exécutable grâce aux objets logiciels…

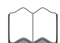
Doc. — C'est bien ce que vise la méthode UML en tout cas ! Elle nous permet de présenter les choses sous forme de « blocs-diagrammes » codifiés aussi représentatifs qu'expressifs. Je dirais qu'une représentation UML constitue le meilleur cahier des charges qu'on puisse désirer.

Cand. — Et un bon dessin vaut mieux que tous les discours, surtout mais pas seulement lors d'une discussion entre informaticiens et non informaticiens !

Doc. — Et si les outils UML sont bien exploités, la simple lecture du plan de bataille permet de relever les incohérences d'un schéma incomplet ou ambigu. Les dépendances entre classes figurent clairement dans les diagrammes. La structure des programmes sera donc mise en évidence au-delà de la conception. L'implémentation de nos objets sera visible sur le plan de leurs interdépendances.

Cand. — Il ne reste pas grand-chose au programmeur avec tout ça !

Doc. — UML favorise pour l'instant la vision globale d'un projet. L'implémentation est contrôlée par des processus encore assez complexes pour mériter les efforts d'un programmeur. La gestion mémoire par exemple… Jusqu'à demain peut-être, où programmer reviendra juste à dessiner quelques diagrammes.



Les trois amigos : Grady Booch, James Rumbaugh, Ivar Jacobson

Il est de coutume d’appeler les trois créateurs d’UML les « trois amigos » dans la communauté informatique. En fait, ces trois auteurs étaient déjà bien connus pour leur apport respectif dans les outils méthodologiques de l’OO, avant qu’ils ne décident de réunir leur force et d’homogénéiser leurs efforts, d’où la présence du terme « unified » dans UML.

Grady Booch depuis 1981, travaillait à la mise au point de formalismes graphiques pour représenter le développement de programmes OO. Il est actuellement le directeur scientifique de la branche « Rational » chez IBM. Sa méthode OOD (Object Oriented Design) fut conçue à la demande du ministère de la Défense des États- Unis. Il avait mis au point des diagrammes de classe et d’objet, des diagrammes d’état-transition, et d’autres diagrammes de processus, pour mieux visualiser les programmes pendant leur exécution. Ces diagrammes devaient accompagner et améliorer l’organisation et la structure de programmes écrits en ADA et C++. Pour l’anecdote, les classes de Booch étaient dessinées comme des espèces de patatoïdes que l’on retrouve si, dans Rational Rose, vous choisissez de dessiner vos diagrammes « à la Booch ».

James Rumbaugh, alors à la General Electric, fut l’auteur d’un formalisme graphique, précédant et en cela anticipant UML, appelé OMT (Object Modelling Technique). C’est d’OMT qu’UML s’est indéniablement le plus inspiré. Cette inspiration fut puisée dans les langages à objet mais également dans la modélisation conceptuelle appliquée à l’analyse et au stockage des données. La plupart des diagrammes importants faisant partie d’UML se retrouvaient déjà dans OMT, comme le diagramme de classe et autres diagrammes dynamiques et fonctionnels. C’est de fait Rumbaugh qui, chez IBM jusqu’en 2006 mais retraité depuis, est le plus impliqué dans la maintenance et l’évolution de son bébé. Il accorde aussi énormément d’importance, au-delà de l’utilisation de ces diagrammes, au suivi d’une méthodologie décomposée en plusieurs phases, de l’analyse à l’implémentation, phases qui, au lieu d’être complètement séquentielles, se recouvrent en partie. Un développement logiciel devrait plutôt se dérouler comme une succession de petites itérations, de courte durée, toutes intégrant de l’analyse et du développement, mais le poids de l’analyse et du développement s’inversant graduellement vers la fin. On retrouve ces directives méthodologiques dans RUP (Rational Unified Process) et dans la programmation agile.

Ivar Jacobson, auteur de la méthode OOSE, Object Oriented Software Engineering, est surtout connu pour avoir recentré l’analyse et le développement informatique sur les besoins humains et, en conséquence, pour l’apport dans UML de la notion de cas d’utilisation et du diagramme correspondant. Il s’est toujours intéressé à la nécessité première d’une bonne représentation du cahier des charges de l’application, ainsi que d’une bonne stratégie de développement, également basée sur une succession de phases courtes, dans lesquelles l’analyse et le développement varient en importance durant la progression du projet. C’est essentiellement à lui que l’on doit RUP, la méthodologie de développement logiciel, qui accompagne souvent l’apprentissage d’UML. Il a crée en 2003 l’entreprise « Ivar Jacobson Consulting ».

Il était bon et très stratégique que ces trois chercheurs et auteurs se rejoignent dans la société Rational (rachetée depuis par IBM, deux des amigos y sont toujours) pour homogénéiser leur notation, car elles circulaient déjà beaucoup dans la communauté, et souffraient de la multiplication de symboles graphiques pour signifier une même chose. Booch le premier à intégrer Rational y attira Rumbaugh puis Jacobson. C’est une des premières raisons à avoir fait le succès et la possible standardisation d’UML : forcer les développeurs à traduire leurs notations favorites dans une notation commune, plutôt que de s’acharner à utiliser de multiples notations concurrentes. Fin 1997, UML est devenu une norme OMG (Object Management Group).

L’OMG est un organisme à but non lucratif, fondé en 1989 sous l’impulsion de grandes sociétés (HP, Sun, Unisys, American Airlines, Philips...). Aujourd’hui, l’OMG fédère plus de 1000 acteurs du monde informatique, y compris Microsoft.et tous les dinosaures informatiques Son rôle est de promouvoir des standards qui garantissent l’interopérabilité entre applications orientées objet, développées dans des environnements informatiques hétérogènes (Windows, Linux, Java, C++). Comme nous le verrons souvent dans le chapitre, UML est une des voies vers cette interopérabilité. Corba (dont nous parlerons dans le chapitre 16 et qui est le deuxième standard de l’OMG) en étant une autre, pour faciliter la communication entre les objets à travers le Web. Ces derniers temps, le dernier cheval de bataille de l’OMG est le MDA (Model Driven Architecture), qui force davantage encore l’utilisation d’UML afin de désolidariser au maximum les développements informatiques des plates-formes sur lesquelles ils se réalisent. Voir UML, en montant encore d’un niveau d’abstraction, comme un possible langage de programmation à part entière s’inscrit totalement dans cette nouvelle croisade de l’OMG.

De mauvaises langues diront que la multiplicité des diagrammes UML, dès lors que certains de ces diagrammes apparaissent vraiment comme étant redondants par rapport à d'autres, provient de l'impossibilité de parvenir à une unification totale des notations, chacun des trois amigos persistant à défendre sa manière de représenter telle ou telle chose. Ils s'en défendent, en répondant que chaque diagramme apporte un nouveau point de vue sur le système, mais, en général, à l'usage, les utilisateurs sont amenés à favoriser deux ou trois de ces diagrammes, dont les plus utilisés : celui de classe et celui de séquence.

Voici quelques ouvrages de référence :

- Une petite entrée en matière très digeste et très économe : *UML*, Martin FOWLER (Campus Press) – traduction anglaise de *UML Distilled*, Second Edition, Addison-Wesley, le livre UML le plus vendu et le plus lu. La nouvelle et troisième édition est un des premiers ouvrages à rendre compte d'UML 2. Pour la petite histoire, Martin Fowler est également très actif pour populariser et promouvoir les méthodologies de développement telle la « programmation agile ». La lecture qui découle de ce livre l'est tout autant.
- Cet ouvrage est un des premiers à rendre compte des modifications apportées par UML 2 : *UML 2 ToolKit*, Hans-Erik ERIKSON, Magnus PENKER, Brian LYONS et David FADO, chez Wiley
- Très didactique car truffé de petits exemples bien pensés : *UML par la pratique – Études de cas et exercices corrigés*, Pascal ROQUES, Eyrolles, qu'il faut choisir, comme toujours, dans sa dernière édition.
- Plus approfondi et plus théorique : *Modélisation objet avec UML*, Pierre-Alain MULLER et Nathalie GAERTNER, Eyrolles.
- Très lié à la manipulation de Rational Rose : *Modélisation UML avec Rational Rose 2000*, Terry QUATRIANI, Eyrolles.
- Très puissant, une référence absolue, car il intègre à la fois UML et les design patterns : *UML et les Design Patterns* de Craig LARMAN chez Campus Press.
- Deux ouvrages introductifs très convaincants tous deux chez Addison-Wesley :
  - *Developing Software with UML*, Bernd OESTERIECH
  - *Using UML*, Perdita STEVENS
- Enfin, une fois n'est pas coutume, un site web très bien fait et très fourni : *uml.free.fr*

Diagrammes UML 2

Nous avons, sans vous aviser de la chose, fait explicitement usage de deux types très particulier de diagramme dans les chapitres qui précèdent. Ces deux diagrammes, appelés *diagramme de classe* et *diagramme de séquence*, font partie des treize diagrammes répertoriés et, surtout, standardisés par le langage graphique de modélisation objet dénommé : UML (Unified Modelling Langage). UML 2 (la deuxième version a été acceptée et standardisée fin 2003) est en effet, depuis quelques années, le standard pour la représentation graphique de

UML 2

UML 2 permet, au moyen de ses 13 diagrammes, de représenter le cahier des charges du projet, les classes et la manière dont elles s'agencent entre elles. Afin d'accompagner le projet tout au long de sa vie, il permet, également, de scruter le programme quand celui-ci s'exécute, soit en suivant les envois de messages, soit en suivant à la trace un objet particulier et ses changements d'état (nous découvrirons mieux le diagramme d'état-transition dans le chapitre 22). Il permet, finalement, d'organiser les fichiers qui constituent le projet, ainsi que de penser leur stockage et leur exécution dans les processeurs. Il y a donc un diagramme pour chaque phase du projet. Certains pensent que le graphisme UML est à ce point puissant qu'il peut servir à la modélisation de n'importe quelle situation complexe (par exemple, le fonctionnement d'un moteur automobile ou des institutions européennes…), que celle-ci se prête ou non, par la suite, à un développement informatique. C'est notamment le point de vue de G. Booch et d'auteurs tels que Martin et Odell de voir dans UML un langage de modélisation qui transcende les seules applications informatiques. Nous restons sceptiques quant à l'exploitation d'UML en dehors du monde informatique et nous nous limiterons dans ce chapitre à l'appréhender comme un moyen de faciliter la conception, le développement et la communication d'un tel projet.

la succession des phases, de l'analyse à l'installation sur site, que comprend un projet informatique. Les diagrammes UML ont pour mission d'accompagner le développement de ce projet, en permettant aux personnes impliquées une autre perception, plus globale, plus intuitive, plus malléable, et plus facilement communicable, de ce qu'ils sont en train d'accomplir. UML est le moyen graphique de garantir que « ce qui se conçoit et se programme bien s'énonce clairement ».

Représentation graphique standardisée

L'avantage d'une représentation graphique standardisée est que tous les développeurs l'abordent, l'appréhendent et la comprennent d'une seule et même manière. Une flèche terminée par une pointe particulière, et reliant deux rectangles, signifiera la même chose, précise au niveau du code, pour tous les programmeurs, mais sera également compréhensible, en partie, pour les personnes, qui, bien qu'impliquées dans le projet, ne mettront pas directement la main à la pâte logicielle.

Ils comprendront suffisamment le sens de cette flèche, et les rectangles qu'elle relie, pour que le code final qui réalisera précisément et définitivement ce que la flèche signifie ne trahisse en rien cette compréhension. Surtout, cela permettra un langage commun, situé quelque part entre deux idiomes, le code final, trop précis et trop peu lisible par tous, et la compréhension intuitive qu'en manifesteront toutes les personnes impliquées, trop imprécise et trop peu formalisée.

Question lisibilité, on conviendra aisément qu'il est beaucoup plus facile de comprendre les interdépendances entre 10 classes lorsque celles-ci sont représentées comme autant de rectangles sur un tableau, un écran ou une page, que de feuilleter un long et pénible listing contenant la définition de ces mêmes classes.

Les diagrammes UML sont formels car ils visent à une sémantique très précise de tous les symboles dont ils sont composés. Cette sémantique fait d'ailleurs l'objet d'un métamodèle UML (« méta- », car écrit lui-même dans les notations UML – essentiellement le diagramme de classe). UML est au code informatique final et exécutable un peu ce que pourrait être la partition musicale à son interprétation ou le projet d'architecture à la maison qui s'ensuivra. Ce projet d'architecture est réalisé à échelle, ou à l'aide de notations que les architectes savent très rigoureuses. Or, ce projet reste compréhensible à vos yeux, même si vous seriez bien en peine de construire votre maison. Les notations utilisées dans le plan d'architecture sont à ce point précises que la maison finale ne devrait en rien trahir ce même plan sur lequel vous vous êtes mis d'accord, dans un monde idéalisé bien sûr (un monde dans lequel, malheureusement, on ne construit pas de maisons…).

En effet, au vu de leur précision sémantique, les diagrammes restent extrêmement contraignants une fois la construction entamée. Très peu de libertés seront laissées au programmeur, une fois ces diagrammes affichés. Ensuite, à l'instar des musiciens et des architectes, qui comprendront leur partition et leur plan d'une seule et même façon, tous les informaticiens comprendront les diagrammes UML d'une seule et même façon, juste avec quelques libertés résiduelles d'interprétation, pour optimiser çà et là une partie du code, non spécifiée dans les diagrammes.

Eu égard à la programmation, UML apparaît, devant la diversité des langages de programmation OO, comme une espèce d'espéranto graphique de ces langages, reprenant tous les mécanismes de base de l'OO, même si leur traduction dans ces langages diffère. Cela permet aux personnes impliquées dans le développement logiciel de restreindre de plus en plus leur apport à la seule conceptualisation et analyse, en se libérant des contraintes syntaxiques propres aux langages et en différant de plus en plus les détails techniques liés à l'implémentation. Grâce à l'ULM (oouhps… pardon !), l'UML, les objets s'envolent. Concrètement, un programmeur C++ et un programmeur Java, C#, Python ou PHP, pourront travailler, pour l'essentiel, dans un langage graphique commun, et automatiser, autant que faire se peut, la traduction des diagrammes dans les différents langages.

Là où un langage choisit d'implémenter l'héritage par `:public` (C++), l'autre par `:` (C#), le troisième par `extends` (Java), le quatrième par `inherits` (VB) et le cinquième par de simples parenthèses (Python) – et encore on en oublie –, UML se borne à le traduire par une flèche pointant de la sous-classe vers la superclasse (voir l'héritage chapitre 11). Quoi de plus clair et de plus compréhensible.

Du tableau noir à l'ordinateur

Ce côté formel et très proche des langages de programmation OO, proximité, comme nous allons le voir, encore accrue dans UML 2, amène la plupart des utilisateurs UML à se répartir sur un axe selon l'importance qu'ils accordent aux diagrammes lors de la réalisation finale du code et l'exigence ou non d'une parfaite fidélité de ce code à ces diagrammes.

À la première extrémité, il y a ceux que nous nommerons les « UMListes du tableau noir », ceux dont le recours à ces diagrammes est moins déterminant et moins contraignant. Ils en reconnaissent l'utilité, surtout lorsque leur programme se complexifie et qu'ils sentent le besoin de petits dessins (comme nous l'avons dit et même pour eux, 10 classes représentées dans un diagramme de classe sur un tableau noir sera toujours plus clair que le code de ces mêmes 10 classes dans un listing) pour s'aider eux-mêmes à résoudre un problème de conception, et surtout pour faciliter la discussion avec leurs collègues programmeurs. Ils y recourent avec parcimonie et la volonté essentielle de se faciliter la vie, sans respecter religieusement tous les détails syntaxiques proposés par UML. Surtout, cela ne les empêche pas, une fois que leur problème est résolu et qu'ils ont pris leurs décisions, de se détourner du tableau noir pour retourner à l'écriture du code. Ils voient surtout les diagrammes UML comme des éléments d'appoint, qu'ils utilisent à des moments précis du développement, lorsque la complexité les dépasse, ou lors d'interactions avec les autres programmeurs. Ils ne sont pas en faveur des environnements de développement logiciel basés sur UML (et de plus en plus fréquents pourtant) car ils leur semblent plus contraignants qu'autre chose. Le code reste leur entière création, leur propriété, leurs soucis, et ils restent très circonspects devant toute production automatisée.

À l'autre extrémité, nous trouvons les « programmeurs UML », ceux qui choisissent d'abord de réaliser ces diagrammes à l'aide d'environnements logiciels tels Together, Rose, Omondo ou Enterprise Architect, environnements plus contraignants, car ils exigent dès le départ une certaine cohérence entre les diagrammes. Par la même occasion, ils se reposent aussi beaucoup sur la génération automatique de code afin d'accompagner dans une large partie leur écriture des programmes. Pour ces derniers, UML doit évoluer vers un langage de programmation à part entière, le but étant de pouvoir programmer un jour en UML tout comme en Java, Python, C# ou PHP, c'est-à-dire de faire d'UML un vrai environnement informatique de conception de programmes exécutables, sans l'intermédiaire, comme c'est le cas aujourd'hui, des codes générés au départ de ces diagrammes. La plupart des environnements de développement permettent cette génération de code dans la plupart des langages OO les plus courants. Mais les concepteurs d'UML ainsi que l'OMG, son organisme de standardisation, veulent aller plus loin dans l'opérationnalisation d'UML. La deuxième version d'UML est la preuve indéniable de cette évolution, comme nous le verrons, par exemple, lorsque nous détaillerons le diagramme de séquence. Celui-ci s'est nettement enrichi dans sa deuxième version afin d'intégrer un maximum de mécanismes procéduraux (test, boucles, …). L'OMG promeut une pratique du développement logiciel dite « MDA » (Model Driven Architecture), c'est-à-dire axée essentiellement sur la modélisation et non plus sur l'écriture de code, laissant cette écriture s'automatiser de plus en plus, grâce à des outils informatiques qui, à partir des diagrammes du modèle, l'adaptent en fonction des plates-formes sur lesquelles le code doit s'exécuter. Plus d'informaticiens mais des modélisateurs, voici ce dont ils rêvent pour l'avenir de la profession.

Nous nous bornons ici à présenter cet axe et ces deux extrêmes. De nombreux praticiens adopteront une attitude intermédiaire. C’est à l’informaticien de voir et à l’avenir de nous dire vers quoi UML évoluera et surtout qu’en feront et quelle importance lui donneront ses utilisateurs et ses partisans.

UML 2 entre le coup de pouce au tableau noir et un vrai langage de programmation

Alors que la plupart des utilisateurs d’UML le voient aujourd’hui comme un complément et une assistance graphique à l’écriture de code, ses créateurs et ses avocats espèrent le voir évoluer vers un véritable langage de programmation, capable d’universaliser et de chapeauter tous ceux qui existent aujourd’hui. En substance, UML pourrait devenir, en lieu et place des langages de programmation, ce que ceux-ci devinrent en remplacement à l’assembleur. Les codes seraient purement et simplement produits par une nouvelle génération de compilateur et les diagrammes s’exécuteraient sans programmation additionnelle. L’informatique n’a de cesse de se caractériser par cette succession de montées en abstraction : portes électroniques, circuits booléens, instructions élémentaires, assembleur, langages de programmation, langage de modélisation (UML).

Programmer par cycles courts en superposant les diagrammes

UML n’est pas une méthodologie, c’est juste un langage. En prenant l’exemple d’une langue étrangère, UML serait plutôt le dictionnaire auquel il manque encore l’Assimil pour mettre la langue en contexte (et qui vous indique vraiment comment trouver votre chemin et demander où sont les toilettes). Rien n’est proposé sur la manière la plus pratique d’utiliser ses diagrammes : lesquels et à quelle étape du développement ? Toutefois, les créateurs, et tout ceux qui font la promotion d’UML en général, accompagnent celle-ci d’une offre en matière méthodologique : le RUP (Rational Unified Process) ou la programmation dite extrême ou agile. Quelques éléments communs à ces nouvelles propositions méthodologiques sont les suivants. Il est capital de travailler par cycle court, cycle reprenant la succession des phases classiques des projets informatiques : cahier de charge, analyse, modélisation, développement, déploiement, et débouchant systématiquement sur un code exécutable tous les mois ou les deux mois au maximum. Le client pour lequel vous développez doit pouvoir rester dans le coup et suivre les progrès. Pour ce faire, rien n’est plus éclairant quant à l’évolution du projet, qu’un programme dont vous lui faites la démonstration. Pas de blabla, de promesses en l’air, mais du concret que diable ! Un programme qui tourne et exécute des choses supplémentaires tous les mois. Le retour du client ainsi que les possibles adaptations de sa demande en seront véritablement facilitées. Les informaticiens se sont rendus compte depuis longtemps que leurs clients n’ont pas toujours les idées très claires sur ce qu’ils veulent réellement au départ du projet et sur les possibilités mirobolantes qui leur seront révélées au fur et à mesure de l’avancement de ce projet.

L’utilisation des diagrammes doit se superposer dans le temps. S’il est évident que ceux qui ont pour objet le cahier des charges du projet seront plus sollicités au début et que ceux qui décrivent l’architecture des fichiers et la manière dont ils s’exécutent sur les machines plutôt à la fin, tous les diagrammes doivent néanmoins pouvoir être repris à tout moment du projet. C’est la raison de cette successsion de cycles qui, chacun, voient toutes les phases classiques d’analyse et de développement se dérouler. Le développement doit montrer une grande flexibilité et capacité d’adaptation (d’où le terme « agile ») et il faut pouvoir revenir sur des décisions, mêmes prises au tout début de la conception si des problèmes imprévus se produisent en fin de parcours (par exemple des problèmes de temps d’exécution). Il est clair que travailler par cycle court aide à cette adaptation, car cela permet tant au client qu’aux développeurs de repérer des problèmes à tout moment, y compris très tôt dans la réalisation.

Diagrammes de classe et diagrammes de séquence

Les deux seuls diagrammes que nous avons utilisés jusqu'à présent sont les diagrammes de classe et les diagrammes de séquence. Pourquoi les avoir utilisés avant d'officiellement vous les présenter ? Parce que nous osons penser qu'ils peuvent parfaitement accompagner l'explication des mécanismes OO, sans nécessiter une explicitation détachée. L'OO les explique au même titre qu'ils servent à expliquer l'OO. Ils permettent une visualisation alternative des mécanismes OO, mais qui aide à la compréhension formelle que l'on doit en avoir.

Le pari que nous avons pris dans les chapitres précédents est que vous ayez compris leur importance et leur signification, sans qu'il n'ait été nécessaire, dans un premier temps, de vous les détailler symbole par symbole. En fait, en plus des différents avantages déjà évoqués, ce pourrait être parmi les meilleurs outils didacticiels pour expliquer l'OO. Une raison simple à cela est que, nonobstant que le langage graphique dans lequel ils s'expriment semble apparemment bien détaché du langage de programmation final, ils restent parfaitement fidèles à leur traduction dans ce langage de programmation.

Certains environnements logiciels de conception UML, comme Together ou Omondo (le plug-in UML d'Eclipse), parmi les plus puissants à ce jour, ont fait de cette traduction automatique leur cheval de bataille et leur valeur ajoutée. Comme nous l'avons dit précédemment, ils évoluent incontestablement dans le bon sens, car de plus en plus d'outils se développent, favorisant l'utilisation de ces diagrammes et assurant, « derrière », la génération automatique de code. Diagramme de classe et diagramme de séquence peuvent se traduire, automatiquement, dans tous les langages de programmation objet, jusqu'à ce qu'un jour cette traduction ne s'avère plus nécessaire, les diagrammes UML se suffisant à eux-mêmes.

À titre de petit exercice, nous allons, dans la suite, nous livrer à une présentation des symboles graphiques les plus usités, propres au diagramme de classe et de séquence, et vous montrer, dans le même temps, les traductions automatiques qui peuvent être faites de ces diagrammes dans les langages C++, C#, Java, Python et PHP 5. Par l'addition d'une fonction `main` (en C++) ou d'une méthode `main` (en Java et C#), nous ferons de ces codes des exécutables qui, lors de l'exécution, produiront à l'écran quelques phrases qui témoignent de leur bon fonctionnement.

Diagramme de classe

Une classe

Commençons par le diagramme de classe. Une classe se décrit par ses trois compartiments : nom, attributs et méthodes.

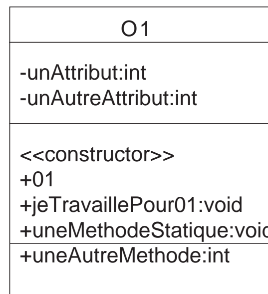


Figure 10-1
Une classe dans le diagramme de classe UML.

En Java : UML1.java

```
class O1 {
    private int unAttribut; // private est indiqué par un signe - dans le diagramme
    private int unAutreAttribut;

       public O1() { // public est indiqué par un signe +
    }
    /* Il serait également possible de générer automatiquement les méthodes d'accès aux attributs
       privés, tels : public void setUnArribut(int unAttribut) et public int getUnAttribut()
       {return unArribut ;} */    public void jeTravaillePourO1() {
        System.out.println ("Je suis au service de toutes les classes");
    }
    public static void uneMethodeStatique() { // la méthode statique est soulignée dans le diagramme
    }
    public int uneAutreMethode(int a) {
         return a;
    }
}
public class UML1 {
    public static void main(String[] args) {
      O1 unObjet = new O1();
       unObjet.jeTravaillePourO1();
    }
}
```

Résultat

```
Je suis au service de toutes les classes
```

En C# : UML1.cs

```
using System;
class O1 {
    private int unAttribut;
    private int unAutreAttribut;

       public O1() {
    }
    /* Il serait également possible de générer les méthodes d'accès qui en C# se définissent comme :
       public int UnAttribut {
          set {
                 unAttribut = value ;
          }
          get {
                 return unAttribut ;
          }
    */
    public void jeTravaillePourO1() {
       Console.WriteLine ("Je suis au service de toutes les classes ");
    }
```

```
    public static void uneMethodeStatique() {
    }
    public int uneAutreMethode(int a){
         return a;
    }
}
public class UML1 {
    public static void Main() {
      O1 unObjet = new O1();
       unObjet.jeTravaillePourO1();
    }
}
```

Résultat

```
Je suis au service de toutes les classes
```

En C++ : UML1.cpp

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
private:
    int unAttribut;
    int unAutreAttribut;
public:
    O1() {
    }
    void jeTravaillePourO1() {
        cout <<" Je suis au service de toutes les classes " << endl;
    }
    void static uneMethodeStatique(){
    }
    int uneAutreMethode(int a){
         return a;
    }
};
int main(int argc, char* argv[]){
    O1* unObjetTas = new O1();  /* un objet construit dans le tas */
    O1 unObjetPile; /* un objet construit sur la pile */
    unObjetTas->jeTravaillePourO1();
    unObjetPile.jeTravaillePourO1();
    return 0;
}
```

Résultat

```
Je suis au service de toutes les classes
Je suis au service de toutes les classes
```

En Python : UML1.py

```
class O1:
    __unAttribut=0
    __unAutreAttribut=0

    def __init__(self):
        pass
    def jeTravaillePourO1(self):
        print ("je suis au service de toutes les classes")
    def uneMethodeStatique():
        pass
    uneMethodeStatique=staticmethod(uneMethodeStatique)
    def uneAutreMethode(self,a):
        return a
unObjet = O1()
unObjet.jeTravaillePourO1()
```

En PHP 5 : UML1.php

```
<html>
<head>
<title> Traduction classe UML </title>
</head>
<body>
<h1> Traduction classe UML </h1>
<br>

<?php
   class O1 {
        private $unAttribut;
        private $unAutreAttribut;
        public function __construct() {

        }
        public function jeTravaillePourO1() {

                print ("je suis au service de toutes les classes <br> \n");
        }
        public static function uneMethodeStatique() {}

        public function uneAutreMethode(int $a) {

            return $a;
        }
   }

   $unO1 = new O1();
   $unO1->jeTravaillePourO1();

?>

</body>
</html>
```

Similitudes et différences entre les langages

À quelques détails de syntaxe près, que le lecteur pourra assez facilement épingler, les codes Java et C# sont équivalents. Il en va un peu différemment des codes Python et PHP 5 qui, comme nous l'avons déjà vu, ne typent ni les attributs ni les méthodes (Python exige de les initialiser ce qui permet en effet de les typer indirectement). En Python, une méthode ne peut se trouver sans instruction d'où la présence de `pass`. Nous avons déjà vu également dans les chapitres précédents la syntaxe particulière de l'encapsulation « private » et des constructeurs. C++, quant à lui, exige de spécifier, lors de la création de l'objet, si cette création se fait sur la mémoire pile ou sur la mémoire tas. De manière générale, C++ offrant bien plus de degrés de liberté que les deux autres, les codes écrits dans ce langage seront toujours moins immédiats à saisir. Les deux possibilités sont illustrées dans le code. Tous les autres langages ne laissent que la mémoire tas pour les objets (C# autorise la pile pour les objets issus des « structures »).

Toujours en C++, si c'est la mémoire tas que nous souhaitons utiliser, il faut que le référent sur l'objet soit explicitement déclaré comme une variable de type adresse, qu'on appelle un pointeur en C++. C'est bien parce qu'il n'y a aucune autre possibilité en Java, Python ou PHP 5 pour la création d'un objet qu'il n'est plus nécessaire de préciser que cette variable est effectivement de type pointeur. C# permet également les deux modes de gestion, mais réserve le tas aux seules classes, et la pile aux structures. En C++, la syntaxe de l'envoi de message sur les deux objets est différente, selon que l'objet est sur la pile ou sur le tas. En fait, l'instruction : `unObjetTas->jeTravaillePourO1()` n'est qu'une réécriture de l'instruction `unObjetTas*.jeTravaillePourO1()`.

Association entre classes

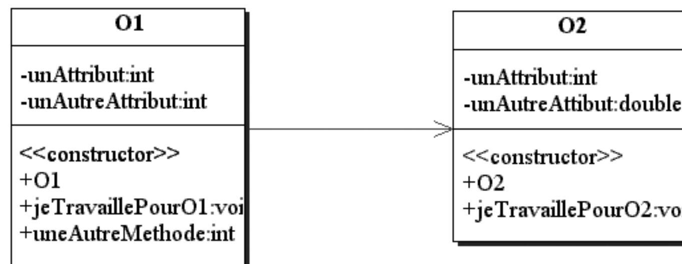


Figure 10-2
Une association dirigée dans le diagramme de classe UML.

Considérons, dans un second temps, une première association « dirigée » entre la classe `O1` et la classe `O2`. Cette relation d'association est celle que nous avons déjà rencontrée entre la proie et le prédateur, la proie et l'eau, etc. La présence de cette association, ici, dans le sens d'`O1` vers `O2` (en l'absence de la flèche, l'association serait considérée bi-directionnelle), exige que, dans le code de la classe `O1`, un message soit envoyé vers `O2`, comme montré dans les cinq codes écrits ci-après, dans les cinq langages.

En Java : UML 2.java

```
class O1 {
    private int unAttribut;
    private int unAutreAttribut ;
    private O2 lienO2; // réalise l'association avec la classe O2
```

```
    public O1(O2 lienO2) /* Le constructeur prévoit de recevoir un référent vers un objet de classe O2  */{
      this.lienO2 = lienO2;
    }
    public void jeTravaillePourO1() {
        lienO2.jeTravaillePourO2() ;   /* Voici l'envoi de message */
    }
    public int uneAutreMethode(int a){
         return a;
    }
}
class O2 {
  private int unAttribut ;
  private double unAutreAttibut ;

  public O2() {}
  public void jeTravaillePourO2() {
    System.out.println("Je suis au service de toutes les classes ") ;
  }
}

public class UML 2{
    public static void main(String[] args){
     O2 unObjet2 = new O2();
     O1 unObjet1 = new O1(unObjet2) ;
/* on passe dans le constructeur de l'objet O1 le référent de l'objet O2 */
    unObjet1.jeTravaillePourO1();
/* un premier message envoyé par le main à l'objet de classe O1 en déclenchera un autre vers un
   ➥ objet de classe O2 */
    }
}
```

Résultat

```
Je suis au service de toutes les classes.
```

En C# : UML 2.cs

```
using System;
class O1 {
    private int unAttribut;
    private int unAutreAttribut;
    private O2 lienO2;

    public O1(O2 lienO2) {
      this.lienO2 = lienO2;
    }
    public void jeTravaillePourO1() {
      lienO2.jeTravaillePourO2();  /* l'envoi de message */
    }
    public int uneAutreMethode(int a) {
      return a;
    }
```

```
}
class O2 {
    private int unAttribut;
    private double unAutreAttibut;
       public O2() {}

    public void jeTravaillePourO2() {
      Console.WriteLine("Je suis au service de toutes les classes");
    }
}
public class UML 2c{
    public static void Main() {
      O2 unObjet2 = new O2();
      O1 unObjet1 = new O1(unObjet2);  /* On passe ici le référent de l'objet O2 lors de la
      ➥ construction de l'objet O1 */
      unObjet1.jeTravaillePourO1();
    }
}
```

Résultat

```
Je suis au service de toutes les classes.
```

En C++ : UML 2.cpp

```
#include "stdafx.h"
#include "iostream.h"
class O2 {
private:
  int unAttribut2;
  double unAutreAttribut2;
public:
  void jeTravaillePourO2() {
     cout << "Je suis au service de toutes les classes" << endl;
  }
};
class O1 {
private:
  int unAttribut;
  int unAutreAttribut;
  O2* lienO2; /* il s'agit d'un pointeur vers un objet de type O2 */
public:
    O1(O2* lienO2) {
        this->lienO2 = lienO2; /* passage du référent */
    }
    void jeTravaillePourO1() {
        lienO2 -> jeTravaillePourO2();  /* l'envoi de message */
    }

    int uneAutreMethode(int a) {
         return a;
    }
};
```

```
int main(int argc, char* argv[]){
    O2* unObjet2Tas = new O2();  /* un objet construit dans le tas */
    O2  unObjet2Pile; /* un objet construit sur la pile */

    O1* unObjet1Tas = new O1(unObjet2Tas);  /* passage du référent */
    O1* unObjet11Tas = new O1(&unObjet2Pile); /* passage du référent de l'objet pile */

    O1 unObjet1Pile(unObjet2Tas);
    O1 unObjet11Pile(&unObjet2Pile);

    unObjet1Tas->jeTravaillePourO1();
    unObjet11Tas->jeTravaillePourO1();
    unObjet1Pile.jeTravaillePourO1();
    unObjet11Pile.jeTravaillePourO1();
    return 0;
}
```

Résultat

```
Je suis au service de toutes les classes.
Je suis au service de toutes les classes.
Je suis au service de toutes les classes.
Je suis au service de toutes les classes.
```

En Python : UML 2.py

```
class O1:
      __unAttribut=0
      __unAutreAttribut=0
      __lienO2=None // il faut initialiser le référent à None

      def __init__(self, lienO2):
            self.__lienO2=lienO2
      def jeTravaillePourO1(self):
            self.__lienO2.jeTravaillePourO2()
      def uneAutreMethode(self,a):
            return a

class O2:
      __unAttribut=0
      __unAutreAttribut=0

      def __init__(self):
            pass
      def jeTravaillePourO2(self):
            print "Je suis au service de toutes les classes"

unObjet2=O2()
unObjet1=O1(unObjet2)
unObjet1.jeTravaillePourO1()
```

En PHP 5 : UML2.php

```
<html>
<head>
<title> Traduction classe UML </title>
</head>
<body>
<h1> Traduction classe UML </h1>
<br>

<?php
   class O1 {
          private $unAttribut;
          private $unAutreAttribut;
          private $lienO2;

          public function __construct($lienO2) {
                   $this->lienO2 = $lienO2;
          }

          public function jeTravaillePourO1() {
                   $this->lienO2->jeTravaillePourO2();
          }
    public static function uneMethodeStatique() {}

          public function uneAutreMethode(int $a) {
                   return $a;
          }
    }

class O2 {
    private $unAttribut;
    private $unAutreAttribut;

    public function __construct() {}

    public function jeTravaillePourO2() {
          print("je suis au service de toutes les classes <br> \n");
    }
}

   $unO2 = new O2();
   $unO1 = new O1($unO2);
   $unO1->jeTravaillePourO1();

   ?>

</body>
</html>
```

Similitudes et différences entre les langages

De nouveau, Java et C# sont quasi équivalents. Les codes Python et PHP 5 ne devraient pas, eux non plus, poser de grands problèmes de compréhension. Une fois encore, l'illustration des deux modes de gestion mémoire en C++ rend le code plus compliqué. Différentes possibilités sont indiquées, selon que l'on utilise des objets créés en mémoire pile ou en mémoire tas. Le caractère « & » signifie que l'on va chercher l'adresse

de la variable plutôt que sa valeur. Lorsqu'il s'agit d'un objet en mémoire pile, l'explicitation de son adresse se fait à l'aide de ce caractère. En revanche, pour un objet en mémoire tas, le référent est directement l'adresse. Le résultat du code C++, malgré les différents modes de gestion de mémoire, exécutera quatre fois le même message et imprimera à l'écran quatre fois la même phrase. Nous avons voulu simplement distinguer quatre possibilités, selon que l'objet `O1` et l'objet `O2` se trouvent dans la mémoire pile ou dans la mémoire tas.

Pas d'association sans message

Il n'y aura jamais lieu de dessiner une telle association dirigée entre deux classes si, nulle part dans le code de la première, n'apparaît un message à destination de la seconde. C'est cela dont permet, également, de s'assurer certains environnements de développements UML, comme Rational Rose. Dans le diagramme de classe et de séquence apparaissant, ci-après, dans Rational Rose, on peut voir que les messages proposés dans le diagramme de séquence sont, en effet, les seules méthodes déclarées dans le diagramme de classe.

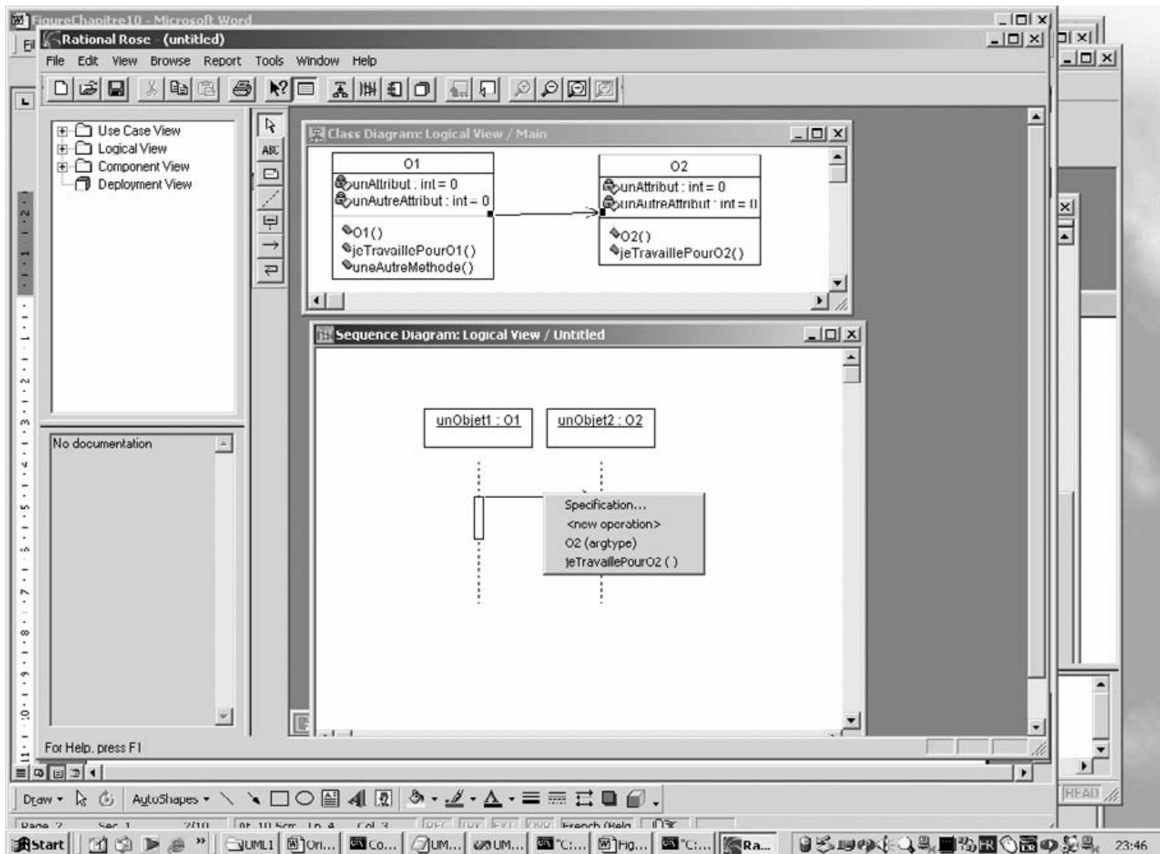


Figure 10-3

Le petit menu apparaissant dans le diagramme de séquence en dessous de la flèche de l'envoi de message reprend les seules méthodes déclarées dans la classe O2.

Si on spécifie un nouveau message, une méthode correspondante viendra automatiquement se rajouter dans la classe qui reçoit le message. Un autre avantage certain de l'utilisation d'un logiciel de développement UML est qu'au-delà de l'assistance graphique, une mise en cohérence automatisée est assurée entres les différents diagrammes. Rational Rose fut le premier environnement logiciel à assurer cette cohérence entre les diagrammes UML et à montrer, ainsi, les avantages d'un logiciel de développement sur la simple utilisation d'un papier et d'un crayon ou du tableau noir. Cette mise en correspondance entre les différents diagrammes est possible grâce au recours à un métamodèle de tous les diagrammes UML réalisé à l'aide du diagramme de classe.

> **Association entre classes**
>
> Il y a association entre deux classes, dirigée ou non, lorsqu'une des deux classes sert de type à un attribut de l'autre, et que dans le code de cette dernière apparaît un envoi de message vers la première. Sans cet envoi de message, point n'est besoin d'association. Plus simplement encore, on peut se représenter l'association comme un « tube à message » fonctionnant dans un sens ou dans les deux.

Rôles et cardinalité

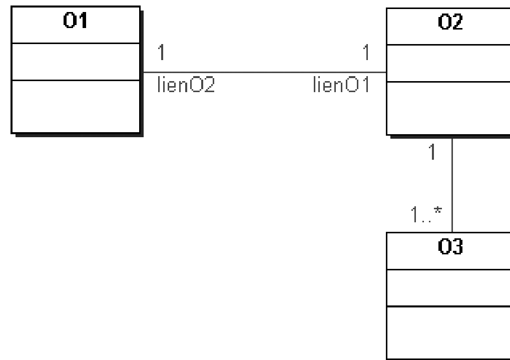


Figure 10-4.
Illustration des rôles et de la cardinalité des associations.

Comme la figure 10-4 le montre, d'autres informations peuvent figurer sur un diagramme de classe UML, afin de caractériser plus finement l'association entre les classes. Dans cette figure, les relations sont toutes bi-directionnelles. Sachant les difficultés que cela pose pour certains langages, nous nous limiterons à la traduction en Java et C++ (en C#, c'est tout à fait identique). Les « rôles » sont les noms donnés, aux deux pôles de l'association, à la classe qui recevra le message par la classe qui lui envoie. Ainsi, le code Java des classes `O1` et `O2` qui est généré à partir de ce diagramme UML pourrait être le suivant , le nom des rôles se substituant au nom des attributs référents .

```
class O1{
  O2 lienO2 ;
}
class O2{
  O1 lienO1 ;
}
```

Une information de type « cardinalité » peut également se retrouver aux deux pôles de l'association, signifiant le nombre d'instances de la première classe en interaction avec le nombre d'instances de la seconde. Cette cardinalité peut rester imprécise, comme dans la figure 10-4, ou plus précise, par exemple « 1…3 », si l'on considère qu'il n'y aura que de un à trois objets entrant dans une interaction avec un autre. Dans la figure 10-4, chaque objet `O2` sera associé à un certain nombre, non défini ici, d'objets `O3` (on verra qu'il est possible d'utiliser un diagramme d'objet de manière à préciser la nature des objets repris par cette cardinalité). Le code Java associé transformera son référent `O3` en un tableau de référents :

```
class O2 {
  O3[] lesLiensO3 ;
// ou ArrayList<O3> lesLiensO3 ;
}
```

Il est possible, depuis les dernières version de Java et de C#, d'utiliser une liste extensible (une `ArrayList`) typée par la classe des objets que cette liste contiendra. L'utilisation d'un tableau en général exige de connaître le nombre d'éléments que celui-ci contiendra.

En C++, comme le pointeur d'un tableau ne pointe toujours que sur le premier élément du tableau, dans les deux cas de figure, une relation `1-1` ou une relation `1-n`, le code sera équivalent (d'où une synchronisation plus délicate entre le diagramme de classe et le code dans le cas de l'utilisation du C++) :

```
class O2 {
  O3* lesLiensO3 ;
}
```

Afin de préciser davantage les problèmes posés par la cardinalité, examinons le petit diagramme UML de la figure 10-5, dans lequel figure une relation, non dirigée cette fois, de type `1 - n`.

Figure 10-5.
Un petit exemple de cardinalité qui peut s'écrire indifféremment « 1-n » ou « 1.. ».*

Utilisons dans la traduction de ce diagramme de classe, une `ArrayList` qui nécessite d'importer dans le code Java le package `java.util`. Il suffit d'actionner la méthode `add` pour pouvoir y ajouter autant d'objets que l'on veut. Nous nous préoccupons dans le code qui suit (et qui également pourrait faire l'objet d'une génération automatique) d'assurer une certaine cohérence dans la relation `1 - n` ; c'est-à-dire que chaque fois qu'un objet est ajouté du côté du « *n* » , on se débrouille pour que celui du côté du « 1 » soit bien celui auquel on vient d'ajouter cet objet, et pas un autre. Si cela vous paraît un peu biscornu, on vous comprend, mais lisez bien le petit code qui suit et vous devriez mieux comprendre.

```
import java.util.*;

class Musicien {
  private ArrayList<Instrument>mesInstruments = new ArrayList(); // déclaration de la liste typée
  public Musicien() {}
```

```
  public void addInstrument(Instrument unInstrument) {
      mesInstruments.add(unInstrument); // ajout d'un élément à la liste
      unInstrument.setMusicien(this); // assure la cohérence de la relation 1-n
                                                   // this réfère l'objet lui-même
  }
}

class Instrument {
   private Musicien monMusicien;
   public Instrument() {}
   public void setMusicien(Musicien monMusicien) {
      this.monMusicien = monMusicien;
   }
}
```

Le code Python qui suit est encore plus détaillé afin de montrer comment la relation `1-n` est assurée et comment la coder. L'autre raison de la présence de ce code est de rendre un petit hommage à la simplicité avec laquelle Python permet au programmeur d'utiliser deux types de données extrêmement puissants (et sur lesquelles nous reviendrons) qui sont les « listes » et les « dictionnaires ». C'est une des forces de ce langage. Une liste est une séquence ordonnée et modifiable d'éléments extrêmement facile à manipuler et à gérer. Un dictionnaire est une collection quelconque d'objets, cette fois-ci non ordonnée, les objets étant indicés par des valeurs arbitraires appelées clés, et, là aussi, fort efficace à l'emploi. Il s'agit certainement du type intégré de données le moins contraingnant et le plus apprécié des programmeurs Python.

```
class Musicien:
    __mesInstruments = [] #déclaration de la liste, difficile de faire plus simple
    __nom = None

    def __init__(self,nom):
        self.__nom = nom
    def addInstrument(self,unInstrument):
        self.__mesInstruments.append(unInstrument) # ajout d'un élément à la liste
        unInstrument.setMusicien(self)
    def printInstrument(self):
        for x in self.__mesInstruments: #instruction très élégante également
            print x
    def __str__(self):
        return self.__nom #définit ce qui apparaît quand on appele le référent de l'objet
class Instrument:
    __monMusicien = None
    __type = None

    def __init__(self,type):
        self.__type = type
    def setMusicien(self,monMusicien):
        self.__monMusicien = monMusicien
```

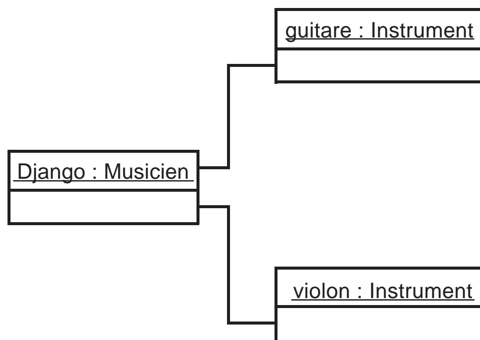
```
    def printMusicien(self):
        print self.__monMusicien
    def __str__(self):
        return self.__type
guitare = Instrument("guitare")
django = Musicien("django")
django.addInstrument(guitare)
django.printInstrument()
guitare.printMusicien()
```

Résultat

```
guitare

django
```

Il est possible de préciser ou de désambiguïser cette cardinalité en recourant à un troisième diagramme UML, très rarement utilisé, sinon à cela, : le *diagramme d'objet*, qui s'apparente à une version instanciée du diagramme de classe. Rappelons que lors de l'exécution du programme, ce sont les objets qui occupent la mémoire pour s'occuper également des traitements. Néanmoins, comme tout ce qu'ils font, y compris les envois de message, est repris dans leur classe, le diagramme de classe suffit le plus souvent à décrire leur comportement. Dans ce diagramme d'objet, en lieu et place des classes, chaque objet apparaîtra, ainsi que le ou les objets avec lesquels il se trouve en interaction. Par exemple, le diagramme d'objet représenté par la figure 10-6 permet de préciser le diagramme de classe dans le cas d'un musicien, dont le référent est `Django`, ne possédant que deux instruments, dont les référents sont `guitare` et `violon`. Étant donnée la présence du `n` dans la cardinalité, il serait tout à fait imaginable que d'autres musiciens possèdent 1 ou 10 000 instruments.

Figure 10-6.
Diagramme d'objet précisant un diagramme de classe.



Finalement, un diagramme de classe peut marquer la différence entre plusieurs référents pointant pourtant vers une même classe, comme illustré par la figure 10-7 en présence du code Java correspondant.

Figure 10-7.
Un diagramme de classe avec deux liens d'association.

```
class Musicien {
      Instrument unPremierInstrument ;
      Instrument unDeuxièmeInstrument ;
}
```

La multiplicité des référents plutôt qu'une même association avec une cardinalité multiple (dans le cas présent, une relation 1-2 pourrait sembler faire l'affaire) tient au rôle différent que sont appelés à jouer les deux référents. Ainsi, le premier instrument pourrait être un instrument solo que le musicien utilise pour l'essentiel et le deuxième, un instrument d'accompagnement, nettement moins sollicité. Les messages qui leur seront envoyés seront suffisamment différents pour qu'il en devienne nécessaire d'en faire deux attributs séparés.

Dépendance entre classes

Nous avons expliqué dans le chapitre 4 qu'il suffit, pour que le lien d'association s'affaiblisse en un lien de dépendance (dans le diagramme de classe, le trait d'association se transforme alors en un trait en pointillés), que la méthode `jeTravaillePourO1(O2 lienO2)` reçoive en argument un objet de type O2. Une autre version que nous avons vue d'un lien de dépendance est indiquée en Java dans le code qui suit :

```
class O1 {
    private int unAttribut;
    private int unAutreAttribut ;

    public void jeTravaillePourO1() {
        O2 lienO2 = new O2() ; /* création d'un objet local O2 */

        lienO2.jeTravaillePourO2() ;   /* Voici l'envoi de message */
    } // l'objet lienO2 disparaît
    public int uneAutreMethode(int a){
        return a;
    }
}
```

Dans cette seconde version d'un lien de dépendance, l'objet O2, qui exécutera le message et justifie la dépendance, n'aura, comme dans le cas précédent, d'existence que pendant l'exécution de la méthode où cet objet est créé. En dehors de cette méthode, l'objet O2, ainsi que le lien entre les deux classes, s'effacent. Dans le cas d'un passage par argument, seul le lien s'efface car l'objet O2 continue à exister. En UML, un lien de dépendance entre deux éléments signifie, simplement, que toute modification dans l'un risque d'entraîner une modification de celui qui en dépend. Comme pour une association, un envoi de messages entre les deux classes pourra avoir lieu. Cependant, ce lien ne dure que le temps de l'exécution de la méthode, et ne s'apparente plus à une propriété structurelle de la classe O1. Ce lien de dépendance entre classes est, en conséquence, moins fréquemment rencontré que le lien, permanent et plus effectif, d'association.

Comme illustré dans la figure 10-8, on retrouve ce même lien de dépendance (une flèche) dans un quatrième diagramme UML, *le diagramme de composants*, entre les fichiers dans lesquels sont stockés des éléments logiciels dépendants. On comprend mieux encore dans le cas des fichiers la nature du lien de dépendance pointillée. En effet, lorsque l'on modifie un fichier, il est fréquent qu'il faille modifier tous ceux qui en dépendent. Par exemple, si l'on modifie le contenu d'une base de données, il faudra également vérifier que tous les programmes qui s'interfacent avec cette dernière soient toujours valides.

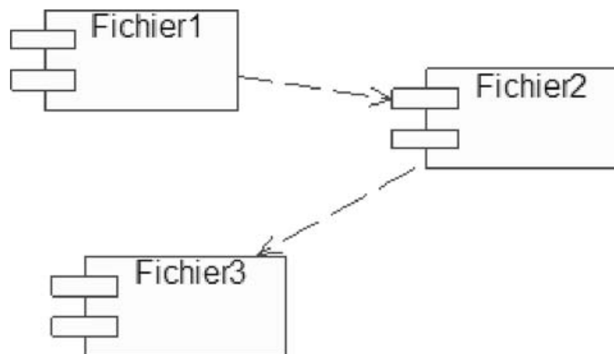


Figure 10-8.
Diagramme de composants reprenant simplement les fichiers et les liens de dépendance entre ceux-ci. Le fichier 1 dépend du fichier 2 et ce dernier dépend du fichier 3.

Composition

Transformons maintenant le lien d’association entre les deux classes en un lien de composition (dit encore d’agrégation forte ou d’agrégation par valeur), et observons-en les conséquences sur le code. Le lien de composition entre deux classes entraîne les instances correspondantes à s’imbriquer l’une dans l’autre dans la mémoire, pile ou tas. La disparition du composite entraînera systématiquement la disparition des composants, et la cardinalité ne pourra prendre que la valeur « 1 » du côté du composite. Le lien d’agrégation faible, dit simplement d’agrégation, ne se comporte pas, une fois traduit en code, de manière différente au lien d’association. Dès lors, parler d’agrégation plutôt que d’association revient simplement à particulariser la sémantique de cette relation et à augmenter la fidélité à la réalité qu’elle dépeint. Si les objets sont physiquement imbriqués les uns dans les autres, comme les atomes dans une molécule, on choisira le lien de composition. Sinon, et tant que subsiste malgré tout une situation d’appartenance entre deux objets, on parlera d’agrégation.

Ainsi, on peut dire d’un enfant qu’il est agrégé dans une famille, mais qu’il est simplement associé à son père. Il aura été, pendant une première période de son existence, délicate à estimer très précisément (car les progrès de la science font qu’elle diminue chaque jour un peu plus), un objet « composant » de sa mère (jusqu’au jour où l’enfant naîtra, et ne le sera plus du tout). On ne sera pas trop surpris d’apprendre qu’une et une seule mère peut porter l’enfant. Les « mères porteuses », c’est autre chose. Ce lien de composition est celui, dans l’écosystème, qui relie la vision à la proie et au prédateur. En effet, les deux animaux sont bien dotés d’une vision, qui les suivra dans la vie comme dans la mort. Cette vision fait partie intégrante d’eux mêmes.

Pour un adepte de la bricole informatique, les composants hardware, comme le processeur, le disque dur ou la RAM, sont agrégés dans l’ordinateur. Pour tous les autres, et ils sont nombreux, ce sont des composants de l’ordinateur qui l’accompagneront d’office à la poubelle. Lors de l’agrégation, la classe « agrégeante » peut indiquer une multiplicité supérieure à 1, alors que lors d’une composition, la classe « composite » ne peut indiquer qu’une multiplicité inférieure ou égale à 1, pour la simple raison qu’un objet ne peut s’imbriquer physiquement dans deux objets à la fois. Le lien de composition peut d’ailleurs se visualiser dans un diagramme d’objet comme un rectangle dans un autre. Éliminons le premier rectangle et celui qui se trouve à l’intérieur disparaît automatiquement. Il n’est pas possible pour le rectangle intérieur d’être présent dans deux rectangles à la fois. Transformons le diagramme UML précédent en sa nouvelle version, dans laquelle la première association se transforme en composition et la deuxième en agrégation, et voyons l’effet résultant dans les différents langages de programmation. De manière à différencier les mécanismes de vie et de mort des objets découlant de ces différents types de relation, dans l’exemple qui suit, la classe `O1` sera reliée à la classe

`O3` par un lien d'agrégation, et à la classe `O2` par un lien de composition (le losange est vide pour l'agrégation et plein pour la composition). Attention à l'emplacement du losange du côté de la classe contenante et non contenue.

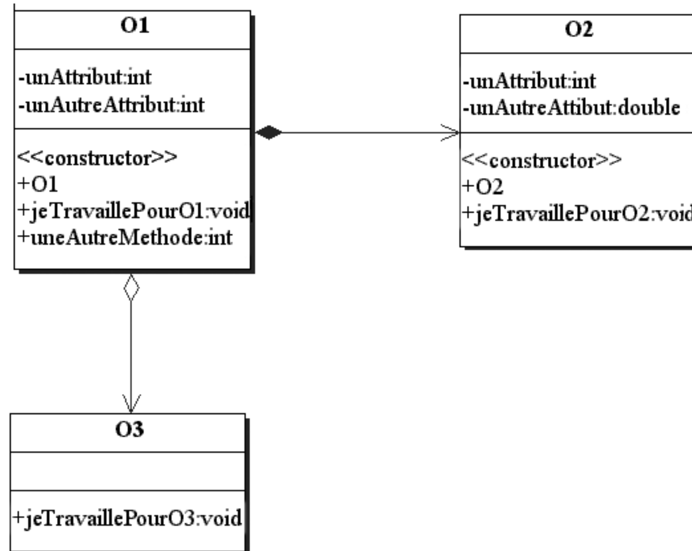


Figure 10-9.
Dans ce diagramme de classe UML, la classe O2 est reliée à la classe O1 par un lien de composition, et la classe O3 est, quant à elle, reliée à la classe O1 par un lien d'agrégation. Un objet O2 sera physiquement intégré dans un objet O1.

En Java

Nous allons, d'abord en Java, proposer deux versions de code réalisant cette relation de composition. Dans le premier code (UML3.java), l'objet `O2` est un composant de `O1`, pour la simple raison qu'il n'a d'existence qu'à l'intérieur de `O1`. En effet, il est construit dans le constructeur d'`O1`, avec, pour conséquence, que le seul référent à cet objet `O2` est un attribut de `O1`. De manière à illustrer cette extrême dépendance entre l'objet `O1` et l'objet `O2`, nous avons recourt à une méthode particulière appelée `finalize()`.

Nous avons déjà rencontré cette méthode, appelée le « destructeur », dans le chapitre précédent. Elle permet, juste avant l'élimination d'un objet, de s'assurer que les ressources utilisables, uniquement à partir de cet objet, seront libérées également. Elle ne peut recevoir d'argument et est d'office sans « retour », vu son mode d'appel. En pratique, il pourrait s'agir d'une connexion à un fichier (la méthode s'occupera donc de libérer l'accès à ce fichier), ou d'une connexion réseau que, là encore, la méthode pourrait interrompre, après avoir éliminé le seul objet connecté au réseau. Ici, nous l'utiliserons, juste pour qu'elle nous indique à quel moment l'objet est éliminé.

Le code qui suit crée un ensemble d'objets à répétition, de manière que le ramasse-miettes soit appelé automatiquement, et récupère un certain nombre de ces objets devenus inutiles. À la différence des codes présentés au chapitre précédent, il n'y a pas, ici, d'appel explicite au ramasse-miettes. Nous préférons l'utiliser comme il l'est dans la plupart des cas, c'est-à-dire, décidant seul de la nécessité de son intervention, quand la mémoire commence à se saturer. Nous forçons simplement son intervention par l'utilisation de la boucle. Le résultat de la simulation des deux codes Java est indiqué juste en dessous de ces codes. Nous n'en montrons qu'une partie, vu la présence de la boucle allant jusqu'à 10 000.

UML3.java

```
class O1 {
    private int unAttribut;
    private int unAutreAttribut;
    private O2 lienO2;
    private O3 lienO3;

    public O1(O3 lienO3) {
      lienO2 = new O2();   /* un lien de composition est créé */
      this.lienO3 = lienO3; /* un lien d'agrégation est créé */
    }
    public void jeTravaillePourO1() {
       lienO2.jeTravaillePourO2();  /* un message vers O2 */
       lienO3.jeTravaillePourO3();  /* un message vers O3 */
    }
    public int uneAutreMethode(int a) {
         return a;
    }

    protected void finalize() /* appel de cette méthode quand l'objet est effacé de la mémoire */{
       System.out.println("aaahhhh... un Objet O1 se meurt  ...");
    }
}

class O2 {
  private int unAttribut;
  private double unAutreAttibut;

  public O2() {}
  public void jeTravaillePourO2() {
    System.out.println("Je suis une instance d'O2 " +
          "au service de toutes les classes");
  }
  protected void finalize(){
      System.out.println("aaahhhh... un Objet O2 se meurt ....");
  }
}
class O3 {
   public void jeTravaillePourO3() {
    System.out.println("Je suis une instance d'O3 " + "au service de toutes les classes");
   }
   protected void finalize(){
      System.out.println("aaahhhh... un Objet O3 se meurt ....");
   }
}
public class UML3 {
    public static void main(String[] args) {
      O3[] lesObjets3 = new O3[10000];
      for (int i=0; i<10000; i++){
        lesObjets3[i] = new O3();
```

```
            O1 unObjet1 = new O1(lesObjets3[i]); // On passe ici le référent de l'objet O3 à l'objet O1
            unObjet1.jeTravaillePourO1();
            unObjet1 = null; /* Par cette instruction, on cherche à se débarasser de l'objet unObjet1,
            ➥ mais elle n'est pas nécessaire */
          }
        }
    }
```

Résultats

```
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
```

Ce n'est, en effet, qu'une petite partie du résultat affiché. Nous constatons, dans le résultat de la simulation, qu'en assignant les référents des objets `O1` à `null`, le ramasse-miettes fait son boulot, comme prévu, en se débarrassant au fur et à mesure, et à notre insu, des objets `O1` devenus encombrants. Notez que cette affectation à `null` n'est pas requise pour faire disparaître l'objet, étant donné que le même référent `unObjetO1` est partagé par tous les objets `O1`, et que chacun de ces objets ne sera donc référé que le temps d'une itération de la boucle.

À la fin de cette itération, il sera livré en pâture au ramasse-miettes. Mais ce qui nous importe le plus ici, c'est l'élimination des objets `O2`, alors que nulle part dans l'écriture du code nous ne l'avons explicitement ordonné. L'objet `O1`, en disparaissant, entraîne dans sa disparition le seul référent à l'objet `O2`, ce qui permet au ramasse-miettes d'y jeter également son dévolu. Nous constatons, en revanche, que les objets `O3`, juste agrégés qu'ils sont, les veinards, subsisteront, quant à eux, jusqu'à l'arrêt pur et simple du programme.

UML3bis.java

```
class O1 {
    private int unAttribut;
    private int unAutreAttribut;
    private O2 lienO2;
    private O3 lienO3;

    public O1(O3 lienO3) {
     lienO2 = new O2(); /* une relation de composition */
     this.lienO3 = lienO3; /* une relation d'agrégation */
    }
```

```
    public void jeTravaillePourO1() {
        lienO2.jeTravaillePourO2(); /* un message vers O2 */
        lienO3.jeTravaillePourO3(); /* un messaege vers O3 */
    }
    public int uneAutreMethode(int a){
         return a;
    }
    protected void finalize(){
      System.out.println("aaahhhh... un Objet O1 se meurt ....");
      /* appel de cette méthode quand l'objet est effacé de la mémoire */
    }
private class O2 { /* la classe O2 est maintenant déclarée à l'intérieur de O1 */
  private int unAttribut;
  private double unAutreAttibut;

  public O2() {}
  public void jeTravaillePourO2() {
    System.out.println("Je suis une instance d'O2 " +
          "au service de toutes les classes");
  }
  protected void finalize() {
      System.out.println("aaahhhh... un Objet O2 se meurt ....");
  }
 }
}
class O3 {
  public void jeTravaillePourO3() {
    System.out.println("Je suis une instance d'O3 " +
          "au service de toutes les classes");
  }
  protected void finalize(){
      System.out.println("aaahhhh... un Objet O3 se meurt ....");
  }
}
public class UML3bis{
    public static void main(String[] args){
       O3[] lesObjets3 = new O3[10000];
       for (int i=0; i<10000; i++) {
          lesObjets3[i] = new O3();
          O1 unObjet1 = new O1(lesObjets3[i]);  // On passe ici le référent de l'objet O3 à l'objet O1
          unObjet1.jeTravaillePourO1();
          unObjet1 = null; /* Par cette instruction, on cherche à se débarrasser de l'objet
          ➥unObjet1, mais elle n'est pas nécessaire */
      }
    }
}
```

Résultat

Le résultat est en tout point semblable au code précédent, une succession et une alternance de :
aaahhhh... un objet O1 se meurt...

```
aaahhhh... un objet O2 se meurt...
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
```

Le contenu de ce code présente une manière encore plus radicale de rendre totalement dépendant les objets O2 des objets O1. Dans le code UML3bis.java, la classe O2 est déclarée et créée à l'intérieur de la classe O1. La classe O2 devient interne à O1. Ce mécanisme, d'utilisation assez rare en Java et C#, car assez subtil, permet de fortement solidariser les classes O1 et O2. Suite à cette écriture, seule la classe O1 pourra disposer de la classe O2. Le lien de composition entre les objets se renforce en s'étendant au niveau des classes. Dans toutes ses méthodes, la classe O2 pourra utiliser tout ce qui caractérise la classe O1 et *vice versa*.

En C#

Le premier fichier C# est assez semblable au fichier Java, à quelques détails de syntaxe près que nous laisserons de côté pour l'instant. Le plus important est, sans doute, le remplacement de la méthode finalize() par un destructeur écrit « à la C++ ». À l'instar du constructeur, le destructeur porte le même nom que la classe en le faisant précéder du caractère « ~ ». Il ne peut recevoir d'argument et est d'office dans « retour ». Le résultat affiché est le même que celui obtenu par le programme Java. Le second fichier Java pourrait également être repris en C#, en le laissant pratiquement inchangé.

Une solution bien plus intéressante est présentée par le fichier UML3bis.cs, dans lequel à la classe O2 on substitue une « structure » O2. En C#, la structure, bien que se décrivant également à l'aide d'attributs et de méthodes, est différente de la classe à plus d'un titre. Parmi ces différences essentielles, nous avons vu que les structures ne peuvent hériter entre elles. Cependant, la seule différence qui nous intéresse ici plus particulièrement est le mode de stockage auxquels les objets sont astreints.

Alors que les objets instances de la classe O1 et de la classe O3 seront installés dans la mémoire tas, et livrés à la gestion par ramasse-miettes couplée à cette mémoire, les objets instance d'une structure seront, quant à eux, stockés dans la mémoire pile, et livrés, pour leur part, au mode de gestion de mémoire de substitution associé à la pile. Une manière, assez directe donc, de faire d'un objet O2 un objet composite d'un objet O1 est de déclarer O2 comme une structure, et de simplement faire de l'objet O2 un attribut de O1. Il sera de ce fait automatiquement attaché au seul objet O1 et disparaîtra avec celui-ci. Vous constatez, par rapport au code précédent, qu'il n'y a pas lieu de construire celui-ci dans le constructeur d'O1. Il se construit automatiquement avec une instance d'O1.

UML3.cs

```
using System;
class O1 {
    private int unAttribut;
    private int unAutreAttribut;
    private O2 lienO2;
    private O3 lienO3;

    public O1(O3 lienO3) {
      lienO2 = new O2();
      this.lienO3 = lienO3;
    }
    public void jeTravaillePourO1() {
        lienO2.jeTravaillePourO2();
        lienO3.jeTravaillePourO3();
    }
```

```
    public int uneAutreMethode(int a){
         return a;
    }
    ~O1() /* le destructeur en C# semblable à la manière de le définir en C++ sans argument et sans
             ➥ retour*/ {
    Console.WriteLine("aaahhhh... un Objet O1 se meurt ....");
    }
}
class O2 {
  private int unAttribut;
  private double unAutreAttibut;

  public O2() {}
  public void jeTravaillePourO2() {
    Console.WriteLine("Je suis une instance d'O2 " +
          "au service de toutes les classes");
  }
  ~O2(){
    Console.WriteLine("aaahhhh... un Objet O2 se meurt ....");
   }
}
class O3 {
   public void jeTravaillePourO3() {
    Console.WriteLine("Je suis une instance d'O3 " +
          "au service de toutes les classes");
   }
   ~O3(){
    Console.WriteLine("aaahhhh... un Objet O3 se meurt ....");
   }
}
public class UML3 {
    public static void Main(String[] args){
      O3[] lesO3 = new O3[100000];
      for (int i=0; i<100000; i++){
          lesO3[i] = new O3();
          O1 unObjet1 = new O1(lesO3[i]);  /* On passe ici le référent de l'objet3 à l'objet1 */
           unObjet1.jeTravaillePourO1();
           unObjet1 = null; /* pas forcément nécessaire */
      }
    }
}
```

UML3bis.cs

```
using System;
class O1 {
    private int unAttribut;
    private int unAutreAttribut;
    private O2 lienO2;
    private O3 lienO3;

    public O1(O3 lienO3) {
      this.lienO3 = lienO3;
    }
```

```
    public void jeTravaillePour01() {
      lien02.jeTravaillePour02();
      lien03.jeTravaillePour03();
    }
    public int uneAutreMethode(int a) {
        return a;
    }
    ~01() { /* le destructeur en C# semblable à la manière de le définir en C++ sans argument et
    ➥sans retour */
       Console.WriteLine("aaahhhh... un Objet 01 se meurt ....");
    }
}
struct 02 {  /* Nous faisons de 02 une structure plutôt qu'une classe */
   private int unAttribut;
   private double unAutreAttibut;

   public void jeTravaillePour02() {
       Console.WriteLine("Je suis une instance d'02 " + "au service de toutes les classes");
  }
}
class 03 {
   public void jeTravaillePour03() {
    Console.WriteLine("Je suis une instance d'03 " +
          "au service de toutes les classes");
   }
   ~03() {
     Console.WriteLine("aaahhhh... un Objet 03 se meurt ....");
    }
}
public class UML3 {
    public static void Main(String[] args) {
      03[] les03 = new 03[100000];
      for (int i=0; i<100000; i++) {
          les03[i] = new 03();
          01 unObjet1 = new 01(les03[i]);  /* On passe ici le référent de l'objet3 à l'objet1 */
          unObjet1.jeTravaillePour01();
          unObjet1 = null; /* pas forcément nécessaire */
      }
    }
}
```

En C++

En C++, au contraire du Java et du C#, rien n'est prévu pour se débarrasser automatiquement des objets qui squattent la mémoire tas. Aucun ramasse-miettes ne viendra seconder un programmeur défectueux. Vous êtes seul maître à bord et, à ce titre, vous ne pourrez quitter le navire qu'une fois que tous les objets l'auront quitté. Pour cela, une instruction vous est proposée : `delete`, qui permet, effectivement, d'éliminer les objets installés dans la mémoire tas. Dans la mémoire pile, le mécanisme d'effacement est automatique, et se fait par désempilement systématique de ce qui y a été le plus récemment empilé.

Afin de suivre à la trace la disparition des objets, le « destructeur » est utilisé qui, comme en C#, porte le même nom que la classe, avec juste un petit « ~ » qui précède son nom. Nous verrons dans la suite que le rôle qui lui est imparti est beaucoup plus important et sensible que celui en Java, plus marginalisé, du `finalize()`. La raison en est, une fois encore, l'absence du ramasse-miettes en C++.

Dans le code `UML3.cpp` ci-après, l'objet de la classe `O2` est un composant de la classe `O1`, alors que l'objet de la classe `O3` lui est simplement associé. On constate que la différence principale réside dans le mode de déclaration de ces attributs. L'objet `O3` l'est, *via* l'utilisation explicite d'un pointeur, l'objet `O2`, non. Dans la mémoire, l'objet `O2` sera comme installé à l'intérieur de l'objet `O1`, alors que l'objet `O3` sera, comme c'est l'usage en Java et C#, juste référé (ou pointé) par une variable adresse stockée dans l'objet `O1`.

Comme le montre le résultat de l'exécution du code `UML3.cpp`, lors de la destruction d'`O1`, l'objet `O2` sera également détruit, puisque son seul champ d'action est l'objet `O1`. De nouveau, les quatre apparitions des mêmes phrases sont liées aux alternatives mémoire tas et pile que nous expérimentons. Les objets « tas » disparaissent suite à l'utilisation de l'instruction `delete`, alors que les objets « pile » disparaissent à la fin du `main`.

UML3.cpp

```
#include "stdafx.h"
#include "iostream.h"
class O2 {
private:
  int unAttribut2;
  double unAutreAttribut2;
public:
  void jeTravaillePourO2() {
    cout << "Je suis une instante d'O2" <<
      " au service de toutes les classes" << endl;
  }
  ~O2(){
    cout <<"aaahhhh... un Objet O2 se meurt ...." << endl;
  }
};
class O3 {
public:
  void jeTravaillePourO3() {
    cout << "Je suis une instance d'O3" <<
      " au service de toutes les classes" << endl;
  }
  ~O3() {
    cout <<"aaahhhh... un Objet O3 se meurt ...." << endl;
  }
};
class O1 {
private:
  int unAttribut;
  int unAutreAttribut;
  O3* lienO3;
  O2 lienO2;
```

```
public:
  O1(O3* lienO3) {
    this->lienO3 = lienO3;
  }
  void jeTravaillePourO1() {
    lienO2.jeTravaillePourO2();
    lienO3 -> jeTravaillePourO3();
  }
  int uneAutreMethode(int a) {
    return a;
  }
  ~O1(){
    cout <<"aaahhhh... un Objet O1 se meurt ...." << endl;
  }
};
int main(int argc, char* argv[]) {
    O3* unObjet3Tas = new O3();
    O3  unObjet3Pile;

    O1* unObjet1Tas = new O1(unObjet3Tas);
    O1* unObjet11Tas = new O1(&unObjet3Pile);

    O1 unObjet1Pile(unObjet3Tas);
    O1 unObjet11Pile(&unObjet3Pile);

    unObjet1Tas->jeTravaillePourO1();
    unObjet11Tas->jeTravaillePourO1();
    unObjet1Pile.jeTravaillePourO1();
    unObjet11Pile.jeTravaillePourO1();

    delete unObjet1Tas;  /* effacement du premier objet sur le tas */
    delete unObjet11Tas; /* effacement du deuxième objet sur le tas */
    return 0;
} /* tous les objets piles disparaissent */
```

Résultat

```
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
Je suis une instance d'O2 au service de toutes les classes
Je suis une instance d'O3 au service de toutes les classes
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
```

```
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O1 se meurt...
aaahhhh... un objet O2 se meurt...
aaahhhh... un objet O3 se meurt...
```

Y a-t-il moyen de réaliser, comme en Java et en C#, un lien de composition entre deux objets, mais installés dans la mémoire tas cette fois ? Cela est tout à fait possible, comme dans le code `UML3bis.cpp` montré ci- après, mais l'utilisation du destructeur devient alors capitale. Cette élimination ne s'effectuant plus automatiquement, comme en Java et C# grâce au ramasse-miettes, il faudra, lors de la destruction d'un objet `O1`, récrire le destructeur de la classe, de manière qu'il s'occupe également de la liquidation de son protégé.

Le code du destructeur de la classe `O1` se doit maintenant d'inclure l'instruction `delete lienO2` , autrement de nombreux objets `O2` risquent de flotter sans ancrage et en totale perdition dans la mémoire. Cette vigilance accrue de la part du programmeur est une des raisons essentielles de la présence du ramasse-miettes dans les autres langages. L'autre raison étant la présence, de nouveau si la vigilance se relâche, de référents fous, obtenus, à l'inverse du cas précédent, par l'usage, cette fois-ci un peu précipité, de l'instruction `delete()`.

UML3bis.cpp

```
UML3bis.cpp
#include "stdafx.h"
#include "iostream.h"
class O2 {
  private:
    int unAttribut2;
    double unAutreAttribut2;
  public:
    void jeTravaillePourO2(){
      cout << "Je suis une instante d'O2" <<
        " au service de toutes les classes" << endl;
    }
    ~O2(){
      cout <<"aaahhhh... un Objet O2 se meurt ...." << endl;
    }
};
class O3 {
  public:
    void jeTravaillePourO3() {
      cout << "Je suis une instante d'O3" <<
        " au service de toutes les classes" << endl;
    }
    ~O3() {
      cout <<"aaahhhh... un Objet O3 se meurt ...." << endl;
    }
};
class O1 {
  private:
    int unAttribut;
    int unAutreAttribut;
```

```
    O3* lienO3;
    O2* lienO2;
  public:
    O1(O3* lienO3) {
      lienO2 = new O2();
      this->lienO3 = lienO3;
    }
    void jeTravaillePourO1() {
      lienO2->jeTravaillePourO2();
      lienO3->jeTravaillePourO3();
    }
    int uneAutreMethode(int a) {
      return a;
    }
    ~O1(){
      cout <<"aaahhhh... un Objet O1 se meurt ...." << endl;
      delete lienO2;  /* Il faut s'occuper également de l'élimination de l'objet O2 */
    }
};
int main(int argc, char* argv[]){
  O3* unObjet3Tas = new O3();
  O3  unObjet3Pile;

  O1* unObjet1Tas = new O1(unObjet3Tas);
  O1* unObjet11Tas = new O1(&unObjet3Pile);

  O1 unObjet1Pile(unObjet3Tas);
  O1 unObjet11Pile(&unObjet3Pile);

  unObjet1Tas->jeTravaillePourO1();
  unObjet11Tas->jeTravaillePourO1();
  unObjet1Pile.jeTravaillePourO1();
  unObjet11Pile.jeTravaillePourO1();

  delete unObjet1Tas;
  delete unObjet11Tas;
  return 0;
}
```

Grâce à la redéfinition explicite du destructeur dans la classe `O1`, et c'est le rôle premier que celui-ci est appelé à jouer en C++, un vrai lien de composition peut exister entre deux classes, même si leurs instances sont stockées dynamiquement dans la mémoire RAM.

En Python

En Python, rien de particulier, c'est encore le ramasse-miettes qui fait la pluie et le beau temps, et les deux codes que nous présentons sont en tout point semblables aux deux codes Java et C#, le premier avec l'objet créé dans le constructeur de la classe composite, le deuxième par le mécanisme de classes imbriquées. Profitez de cet exemple pour vérifier la souplesse et la facilité d'utilisation des dictionnaires :

UML3.py

```
class O1:
        __unAttribut=0
        __unAutreAttribut=0
        __lienO2=None
        __lienO3=None

        def __init__ (self, lienO3):
                self.__lienO2=O2()
                self.__lienO3=lienO3
        def jeTravaillePourO1(self):
                self.__lienO2.jeTravaillePourO2()
                self.__lienO3.jeTravaillePourO3()
        def uneAutreMethode(a):
                return a
        def __del__(self):
                print "aaahhhh... un Objet O1 se meurt ..."

class O2:
        __unAttribut=0
        __unAutreAttribut=0

        def __init__(self):
                pass
        def jeTravaillePourO2(self):
                print "je suis une instance d'O2 " + "au service de toutes les classes"
        def __del__(self):
                print "aaahhhh... un Objet O2 se meurt....."

class O3:
        def jeTravaillePourO3(self):
                print "je suis une instance d'O3 " + "au service de toutes les classes"
        def __del__(self):
                print "aaahhhh... un Objet O3 se meurt....."

lesObjetsO3={}
i=0
while i<10:
        lesObjetsO3[i]=O3()
        unObjet1=O1(lesObjetsO3[i])
        unObjet1.jeTravaillePourO1()
        unObjet1=None
        i+=1
```

UML3bis.py

```
class O1:
        __unAttribut=0
        __unAutreAttribut=0
        __lienO2=None
        __lienO3=None

        def __init__ (self, lienO3):
                  self.lienO2=self.O2()
                  self.lienO3=lienO3
        def jeTravaillePourO1(self):
                  self.lienO2.jeTravaillePourO2()
                  self.lienO3.jeTravaillePourO3()
        def uneAutreMethode(a):
                  return a
        def __del__(self):
                  print "aaahhhh... un Objet O1 se meurt ..."

        class O2: #la classe O2 est maintenant imbriquée dans la classe O1
                  __unAttribut=0
                  __unAutreAttribut=0

                  def __init__(self):
                        pass
                  def jeTravaillePourO2(self):
                        print "je suis une instance d'O2 " + "au service de toutes les classes"
                  def __del__(self):
                        print "aaahhhh... un Objet O2 se meurt....."

class O3:
        def jeTravaillePourO3(self):
                  print "je suis une instance d'O3 " + "au service de toutes les classes"
        def __del__(self):
                  print "aaahhhh... un Objet O3 se meurt....."

lesObjetsO3={}
i=0
while i<10:
        lesObjetsO3[i]=O3()
        unObjet1=O1(lesObjetsO3[i])
        unObjet1.jeTravaillePourO1()
        unObjet1=None
        i+=1
```

En PHP 5

```
<html>
<head>
<title> Relation de composition </title>
</head>
<body>
<h1> Relation de composition </h1>
```

```
<br>

<?php
   class O1 {
         private $unAttribut;
         private $unAutreAttribut;
         private $lienO2;
      private $lienO3;

         public function __construct($lienO3) {
                 $this->lienO2 = new O2();
                 $this->lienO3 = $lienO3;
         }

         public function jeTravaillePourO1() {
                 $this->lienO2->jeTravaillePourO2();
                 $this->lienO3->jeTravaillePourO3();
         }

         public function uneAutreMethode(int $a) {
                 return $a;
         }

         public function __destruct() {
          print ("aaahhhh.... un objet O1 se meurt .... <br> \n");
         }
   }

   class O2 {
         private $unAttribut;
         private $unAutreAttribut;

         public function __construct() {}

         public function jeTravaillePourO2() {
          print("je suis O2 au service de toutes les classes <br> \n");
         }

         public function __destruct() {
                 print ("aaahhhh.... un objet O2 se meurt .... <br> \n");
         }
}

   class O3 {
         public function jeTravaillePourO3() {
                 print("je suis O3 au service de toutes les classes <br> \n");
         }

         public function __destruct() {
                 print ("aaahhhh.... un objet O3 se meurt .... <br> \n");
         }
}
```

```
    while ($i<10) {
          $lesObjets03[$i]=new O3();
          $unObjet1=new O1($lesObjets03[$i]);
          $unObjet1->jeTravaillePourO1();
          $unObjet1 = NULL;
          $i+=1;
    }
?>

</body>
</html>
```

Rien de bien particulier. Nous ne présentons en PHP 5 que la première version de la compostion car les classes internes ne semblent pas vouloir être supportées par le langage à l'heure où nous écrivons ces lignes. Ici également, le « ramasse-miettes » remplit son rôle en se débarassant des objets inutiles et, cela, dès le moment où ils le sont devenus. Le ramasse-miettes agit à tout moment et non pas de façon intermittente comme en Java et .Net. Ainsi, à chaque nouvel affectation du référent, les objets précédemment référés disparaîtront de la mémoire.

Composition

Bien que le lien d'agrégation et de composition servent à reproduire, tous deux, une relation de type « un tout et ses parties », le lien de composition rend, de surcroît, l'existence des objets tributaires de l'existence de ceux qui les contiennent. L'implantation de cette relation dans les langages de programmation dépendra de la manière très différente dont les langages de programmation gèrent l'occupation mémoire pendant l'exécution d'un programme. Nous retrouvons le besoin pour les programmeurs C++ de redoubler d'attention par l'utilisation du `delete`. Pour les programmeurs des autres langages, ils devront s'assurer que le seul référent de l'objet contenu est possédé par l'objet contenant.

Classe d'association

Une dernière possibilité offerte par le diagramme de classe est la notion de classe d'association, illustrée par la figure 10-10 dans le cas de musiciens et de leurs instruments.

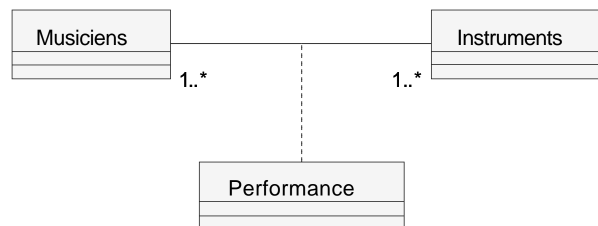


Figure 10-10.

Petit exemple d'une classe d'association : la classe Performance qui relie un et un seul musicien à son instrument pendant la performance.

```
class Performance {
   Musicien unMusicien ;
   Instrument unInstrument ;
}
```

Dans la figure 10-10, la classe `Performance` est une classe d'association qui fait le lien entre un et un seul musicien et un et un seul instrument, et cela bien qu'un musicien puisse être associé à plusieurs instruments et réciproquement. Elle se rattache par un trait pointillé à la liaison entre les deux classes qu'elle associe. Il y aura un objet performance bien particulier pour toute association entre un objet musicien et un objet instrument. Chaque objet de la classe d'association possédera un référent vers un objet particulier de chacune des classes associées. D'autres classes d'association typiques sont l'emprunt d'un livre par un lecteur dans une bibliothèque, la réservation d'un billet de spectacle par un client donné ou l'emploi d'une personne dans une société.

Les paquetages

Il est également possible de représenter les paquetages et leurs liens de dépendance ou d'imbrication, comme le montre la figure 10-11. Un paquetage sera dépendant d'un autre lorsqu'une classe ou un fichier contenu dans le premier s'avère dépendant d'une classe ou d'un fichier contenu dans le deuxième.

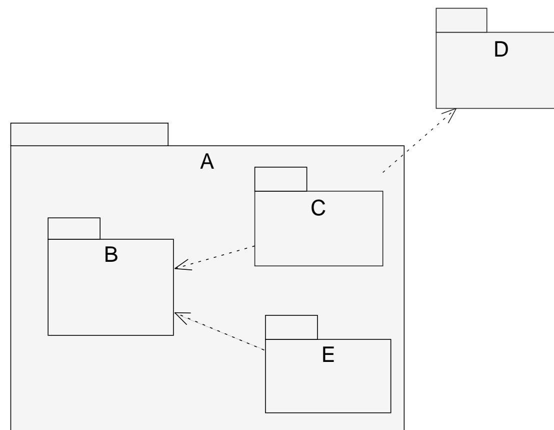


Figure 10-11.
Le paquetage E est dépendant du paquetage B. Le paquetage C est dépendant des paquetages B et D. Les paquetages B, C et E se trouvent à l'intérieur du paquetage A.

Les bienfaits d'UML

De manière à illustrer l'apport précieux des diagrammes de classe, deux exemples sont présentés ci-après. Le premier reprend le petit écosystème du chapitre 3 (dont le code sera esquissé par la suite). Le second, non accompagné d'un code, présente ce que pourrait donner une première ébauche d'analyse, ayant comme but final la réalisation d'un logiciel de match de football. Nous représentons les classes le plus simplement possible, sans y indiquer leurs attributs et méthodes. Ce qui nous intéresse le plus ici, davantage que les classes elles-mêmes, c'est la nature de leurs relations. Ces diagrammes de classe devraient vous apparaître assez compréhensibles, même si nous ne les détaillons pas ici. C'est de fait, dans ses grandes lignes, le pari d'UML.

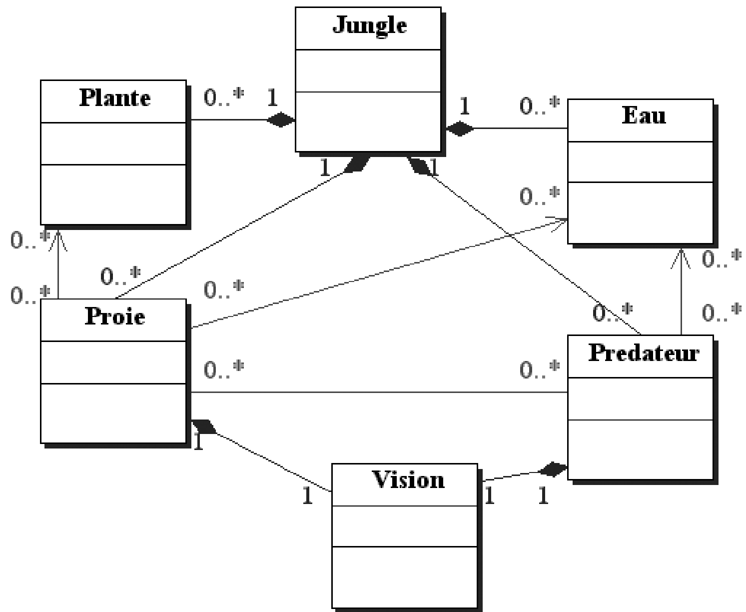
Un premier diagramme de classe de l'écosystème

Voir figure 10.12.

Des joueurs de football qui font leurs classes

Voir figure 10.13.

Figure 10-12
Premier diagramme de classe de l'écosystème du chapitre 3.



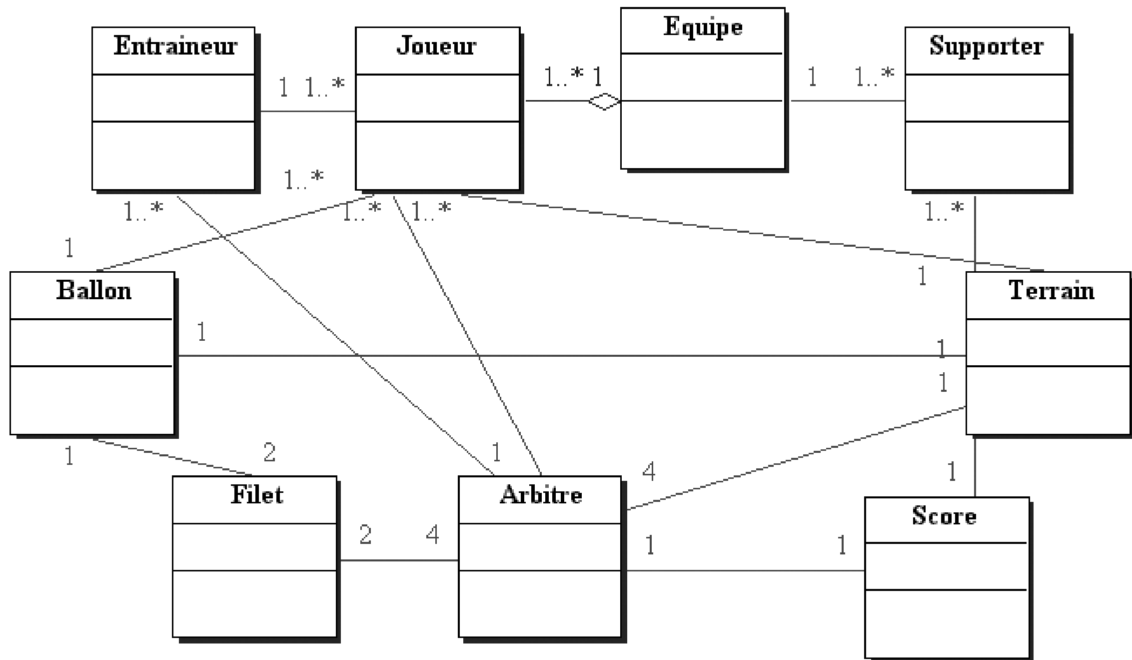


Figure 10-13
Petit diagramme de classe d'une éventuelle simulation d'un match de football.

Les avantages des diagrammes de classe

Plusieurs points importants doivent être soulignés en concluant cette présentation centrée sur le diagramme le plus utilisé des treize diagrammes UML : le diagramme de classe. D'abord, nous sommes loin d'en avoir fini avec l'utilisation de ce dernier. Nous y reviendrons, pas plus tard que dans le prochain chapitre, car nous avons, pour l'instant, mis sous le boisseau un type de relation entre les classes, fondamental en OO : la relation d'héritage. Or, nous voyons bien que, tant dans l'écosystème que lors du match de football, nous pourrions simplifier la conception du modèle, en factorisant dans une classe `Animal` tout ce qu'il y a de commun entre la proie et le prédateur, et en spécialisant en défenseur et attaquant les joueurs de football. Le diagramme de classe permet de représenter ces liens d'héritage et de spécialisation d'une manière qui sera décrite plus avant.

Ensuite, l'isomorphisme entre le diagramme de classe et les codes logiciels qui le traduisent est tel que plusieurs environnements de développement UML, à l'instar de ceux utilisés principalement dans ce livre : TogetherJ et Omondo, permettent une parfaite synchronisation (dénommée en anglais « reverse engineering ») entre ce diagramme et le squelette de code. On parle de squelette de code, car l'intérieur des méthodes n'est nullement spécifié à ce stade. C'est d'ailleurs souvent ce code généré (les principaux langages OO sont concernés), qui permet à un diagramme de classe, créé dans un de ces environnements, d'être récupéré dans un autre.

Cette synchronisation peut être extrêmement précieuse, quand on cherche à homogénéiser les codes développés par des développeurs divers, ainsi qu'à documenter ces développements de manière uniforme. La génération automatique de code, qui tenait préalablement du gadget, est une évolution désirée dans la communauté informatique (comme nous l'avons dit, c'est clairement le chemin tracé par l'OMG avec le MDA), qui préfère consacrer l'essentiel de ses efforts à la conception et l'analyse des logiciels en UML, tout en laissant, chaque jour davantage, aux environnements de développement UML, le soin de générer le code qui concrétise cette conception.

Par ailleurs, et le modèle embryonnaire du match de football est là pour en témoigner, ce diagramme est une aide extrêmement précieuse à la conception du logiciel et à l'interaction entre les développeurs. Il ne faut pas être un informaticien de génie pour le réaliser et le comprendre. Tout amateur de football (même hooligan) en comprendra aisément la structure. Cela explique que l'existence de ces diagrammes facilite grandement l'interaction entre des personnes impliquées dans un projet, personnes qui peuvent intervenir à différents niveaux de la conception : des décideurs aux programmeurs, et dont le goût pour la programmation peut largement varier.

Ils sont une preuve éclatante que l'orienté objet permet au monde qui nous environne d'être la principale source d'inspiration pour le portrait logiciel que l'on cherche à en tirer. L'OO rapproche la programmation du monde réel et, chemin faisant, l'éloigne des instructions élémentaires des processeurs. Le diagramme de classe ouvre la voie à une programmation complètement automatisée, à partir de la seule élicitation des acteurs du problème et des interactions qu'ils entretiennent entre eux. Il est aussi l'ultime étape de cette montée en abstraction qui n'a de cesse de caractériser les développements logiciels.

Finalement, ces diagrammes permettent une appréhension globale du développement, qui est impossible quand seul le code est disponible. Par exemple, dans le chapitre 21 dédié aux graphes informatiques, on verra apparaître la structure récursive d'une des solutions, avec la présence très nette d'une fermeture dans le diagramme. Différentes solutions architecturales et algorithmiques peuvent être rapidement évaluées, et surtout comparées, grâce aux diagrammes de classe. Ces derniers sont devenus incontournables dans des projets informatiques de plus en plus lourds et complexes ; utilisés dès le début, ils les accompagnent du long, et peuvent être discutés à différents stades de leur développement, documentés et uniformisés. De par sa décomposition naturelle en classe, l'OO offre à l'informatique une manière de simplifier ses développements. Les diagrammes de classe accompagnent cette offre, la rendant plus attrayante encore, par son détachement accru de l'écriture logicielle et du fonctionnement intime du processeur.

Un diagramme de classe simple à faire, mais qui décrit une réalité complexe à exécuter

Alors que l'analyse par UML favorise une approche éclatée, classe par classe, au pire une classe se devant de connaître l'interface de quelques autres, l'exécution qui en résulte peut elle, au contraire, impliquer bien plus d'objets, et de manière assez tortueuse. Rien de grave à cela, puisque vous avez laissé la main au seul processeur. Toute la partie compliquée de création, de localisation des objets et de transmission des messages, allant jusqu'au droit de vie et de mort pour chaque objet, lui incombe.

Prenez par exemple le marquage d'un but. Le joueur se limite à taper dans la balle. La balle se limite à se déplacer. Les filets se limitent à constater qu'une balle les « pénètre ». S'ils sont pénétrés, ils le signalent à l'arbitre. L'arbitre, alors, envoie le message `incrémenteScore()` au score. Le score déclenche les cris de joie ou de désespoir des supporters, etc. Nulle part, l'effet très indirect du coup de pied dans le ballon sur les cris des supporters n'a été réellement anticipé, pensé et décortiqué. Cet effet résulte d'une avalanche de messages, circulant de manière conditionnelle (si … alors) et de lien en lien. Ces liens, au cas par cas, sont les seuls à avoir fait l'objet d'une vraie réflexion. Le joueur ne s'occupe que du ballon même si l'effet d'un de ses coups de pied pourrait être les cris des supporters. Le programmeur n'a pas pas programmé cet effet. C'est l'aboutissement d'une succession de messages, chaque programmeur n'ayant pensé et conçu qu'une très petite partie de ceux-ci.

On décompose le problème, on le pense acteur par acteur, même si le jeu d'interaction d'acteurs qui s'ensuit se révèle complexe, au point parfois de vous surprendre. C'est l'image du feu d'artifice que nous avons dans le chapitre 4 empruntée à Bertrand Meyer. La chronologie des messages et des effets de ces derniers n'est jamais attaquée de front. On pense les envois de message et ce qui conditionne ceux-ci, de manière logique, au cas par cas. S'il y a une succession de messages, c'est que l'ensemble des conditions se trouve vérifiée, mais ce déroulement n'aura jamais fait l'objet d'une étude exhaustive préalable. C'est un peu comme des musiciens d'orchestre qui répéteraient, deux par deux, de manière à apprendre à jouer ensemble. Quand ils se trouvent, tous, dans la fosse d'orchestre, pour la première fois, la musique qu'ils produisent individuellement s'harmonise, prend corps. Il ne faut pas programmer les classes comme un tout, en se préoccupant de ce qu'elles peuvent faire pour nous, mais plutôt de les programmer en pensant à ce que celles-ci pourront faire d'elles-mêmes et entre elles. C'est la clé de la pratique OO, sans véritable équivalent dans la pratique procédurale.

Procéder de manière modulaire et incrémentale

La décomposition en modules indépendants permet, non seulement de simplifier le travail d'analyse, mais de faciliter également la progression du projet dans le temps, par rajout progressif de nouveaux modules. Ces modules, pour autant que leur interface et leur implémentation soient clairement tenues séparées, résisteront assez bien à cette incrémentation progressive, et seront réutilisables dans des contextes très différents. Vous pourrez réutiliser la balle au handball, au volley-ball, et même au rugby, si la signature de la méthode responsable du déplacement de la balle a été clairement détachée de l'implémentation de ce déplacement. Cette pratique incrémentale et modulaire, ponctuée de programmes de complexité croissante, est l'essence même des nouvelles méthodologies de développement qui souvent accompagnent la promotion de l'OO.

Diagramme de séquence

Les diagrammes de séquence, que nous avons déjà approchés dans les chapitres 4 et 5, peuvent accompagner le développement d'un projet, à un stade plus avancé que les diagrammes de classe. En effet, ces derniers permettent de visualiser le programme lors de son exécution. Quand celui-ci s'exécute en effet, ce sont les objets

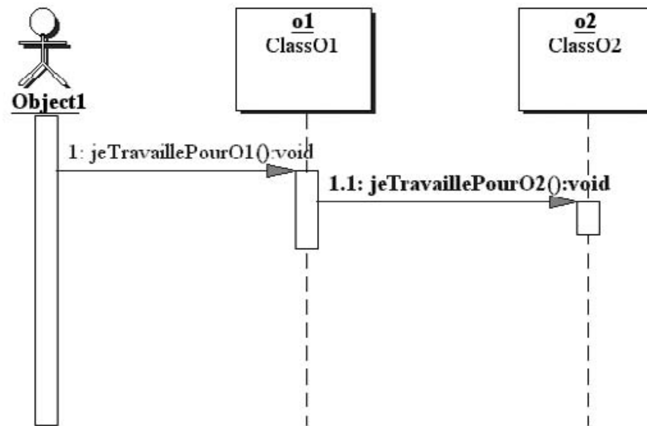


Figure 10-14.
Un petit diagramme de séquence.

qui s'agitent, en se sollicitant mutuellement par l'envoi de messages, et ce sont précisément ces envois de message qui constituent l'essentiel de ces diagrammes. Tous les programmes présentés plus haut, et quel que soit le langage de programmation utilisé, peuvent se représenter par le diagramme repris du chapitre 4, quand l'objet `o1` issu de la classe `O1`, lors de l'exécution de sa méthode `jeTravaillePourO1()`, envoie le message `jeTravaillePourO2()` à l'objet `o2` issu de la classe `O2`.

Le temps s'écoule de haut en bas, la succession des messages aussi. La présence des rectangles ainsi que des numéros de messages (1, 1.1…), indiquent la succession et l'emboîtement des appels de méthodes correspondants. Nous reviendrons sur cet emboîtement par la suite, car il peut être fortement relaxé, quand les programmes disposent pour s'exécuter de plusieurs processeurs ou de plusieurs « threads » en parallèle (le multithreading sera présenté au chapitre 17). Dans un cadre d'exécution de programme fonctionnant uniquement de manière séquentielle, on comprend bien la raison de l'emboîtement des rectangles. Il faut bien que la méthode `jeTravaillePourO2` termine son exécution afin que la méthode `jeTravaillePourO1` puisse reprendre la sienne, d'où le premier rectangle englobant le deuxième et l'addition successive de « . » dans la numérotation des messages.

Ainsi, les flèches qui décochent les messages auront des terminaisons différentes, selon que ces messages sont synchrones (l'expéditeur est bloqué en attendant que le destinataire en ait fini avec sa méthode) ou asynchrones (l'expéditeur et le destinatire peuvent travailler en parallèle). Par défaut, les messages sont considérés comme synchrones, s'exécutant sur un processeur unique et sans multithreading. Dans pareil cas, lorsqu'une des instructions d'un corps d'instructions consiste en l'envoi d'un message vers un autre objet, le flot d'instructions du premier objet, expéditeur du message, s'interrompt, le temps que le flot d'instructions du deuxième objet, destinataire du message, déclenché par l'envoi de message, se termine. Le petit bonhomme dans le diagramme représente le point de départ de la séquence de message. Pour un programme dont on représenterait l'entièreté du diagramme de séquence, le petit bonhomme représenterait le `main`. Cependant, un diagramme de séquence peut démarrer au départ de n'importe quel appel de méthode.

Très tôt, certains environnements de développement UML, les précurseurs (comme TogetherJ) ont tenté, au prix de contorsions importantes, et acceptant quelques écarts par rapport à la norme UML, de synchroniser, aussi parfaitement que possible, l'écriture du logiciel et le diagramme de séquence qui l'accompagne. À titre d'exemple, et sans le commenter davantage, un code et le diagramme de séquence généré automatiquement par TogetherJ à partir de celui-ci sont montrés ci-après. L'observation conjointe du code et du diagramme

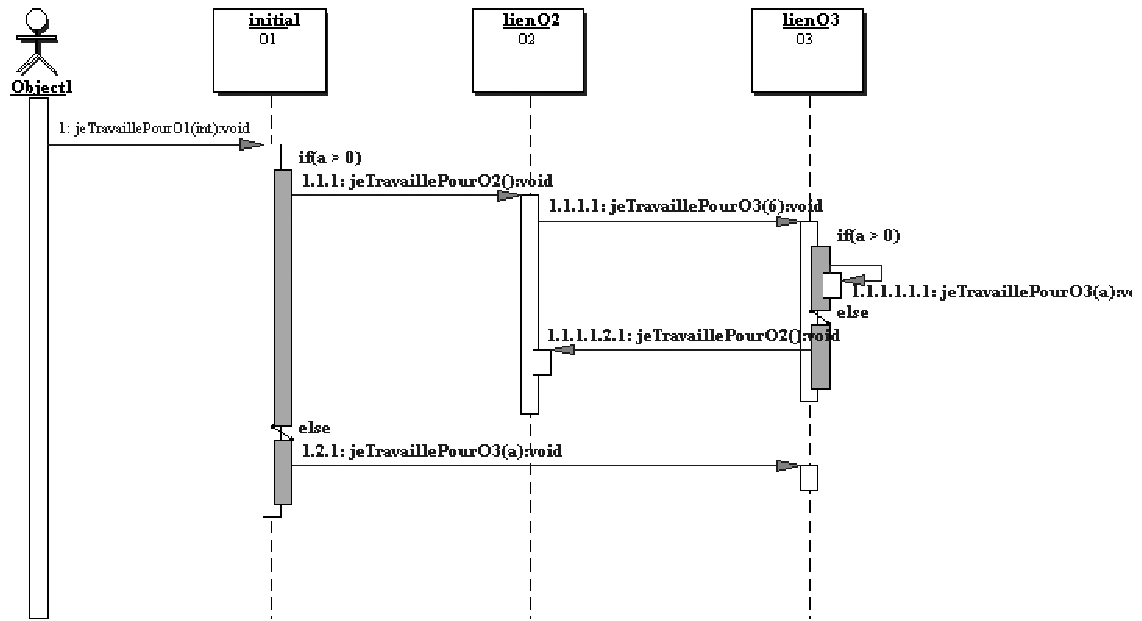


Figure 10-15

Un diagramme de séquence plus compliqué, généré automatiquement à partir du code Java.

résultant devrait permettre d'apprécier l'effort important fourni historiquement par certains développeurs, pour synchroniser, davantage encore, la symbolique des diagrammes UML et l'écriture du logiciel.

```
public class O1 {
  private int attribute1;
  private O2 lienO2;
  private O3 lienO3;

  public void jeTravaillePourO1(int a) {
    if (a > 0){
      lienO2.jeTravaillePourO2();
    }
    else{
      lienO3.jeTravaillePourO3(a);
    }
  }
}
class O2 {
  private O3 lienO3;

  public void jeTravaillePourO2() {
    lienO3.jeTravaillePourO3(6);
  }
}
```

```
class O3 {
  private O2 lienO2;

  public void jeTravaillePourO3(int a) {
    if (a > 0){
      a--;
      jeTravaillePourO3(a);
    }
    else {
      lienO2.jeTravaillePourO2();
    }
  }
}
```

En cela, et vu la nouvelle version d'UML et les nombreuses additions affectant le diagramme de séquence, on peut dire de Together qu'il était en avance sur son temps. En effet, dans UML 2, le diagramme de séquence s'est considérablement enrichi, de façon à synchroniser davantage encore la représentation des diagrammes et le code correspondant. Il ne s'est pas, malheureusement pour eux, enrichi à la manière de Together, mais a conservé le type d'addition que ce dernier avait déjà anticipé dans sa version à lui des diagrammes. Cela va bien sûr dans le sens d'un rhabillage d'UML vers une nouvelle forme de langage de programmation.

Voici un exemple de petit code Java. On supposera que les classes `O2`, `O3` et `O4` existent par ailleurs avec les méthodes appropriées. La figure 10-16 présente la nouvelle mouture du diagramme de séquence UML 2 correspondant, généré cette fois par la nouvelle version de Together ou par Omondo (le logiciel UML qui se greffe sur l'environnement de développement Java d'Eclipse).

```
class O1
{
  private O2 o2;
  private O3 o3;
  private O4 o4;
  public void jeTravaillePourO1() {
        int i = 0;
        int j = 0;
   while (i < 100) {
        if (j > 20) {
             o2.jeTravaillePourO2();
        }  else {
             o3.jeTravaillePourO3();
        }
     i++;
         }
   if (j < 50) {
        o4.jeTravaillePourO4();
         }
   }
}
```

On voit apparaître dans le diagramme de nouveaux éléments graphiques qui permettent un meilleur collage aux mécanismes classiques de la programmation procédurale, qui continuent à constituer le corps des méthodes. Ainsi, les trois grands rectangles intitulés `loop`, `alt` et `opt` correspondent à des « frames », des régions du

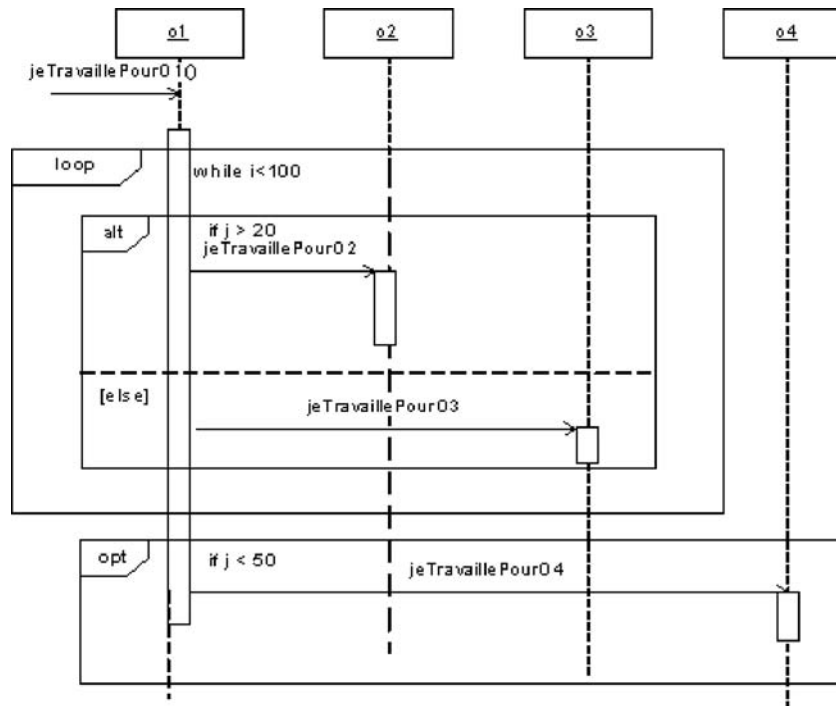


Figure 10-16
Nouveau diagramme de séquence dans UML 2.

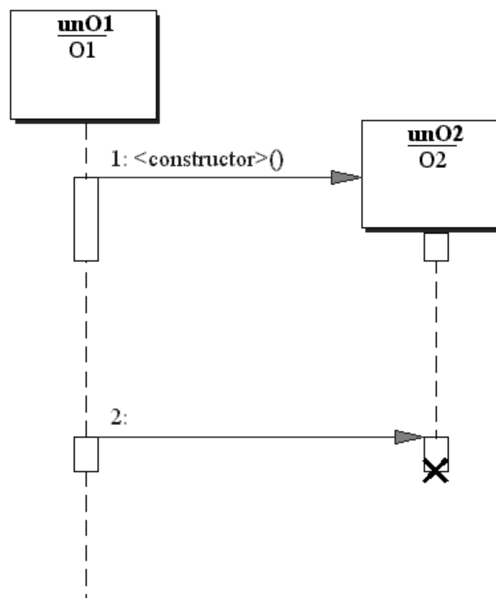
diagramme de séquence divisées en un ou plusieurs fragments (comme dans le cas du frame `alt` divisé en deux fragments d'un `if - else`). Une fois labellées par le petit texte en haut à gauche du frame, ces parties du diagramme de séquence peuvent se retrouver n'importe où dans un autre diagramme de séquence. Cela permet, par exemple, de découper le diagramme de séquence en parties distinctes et de déplacer ces parties d'un diagramme à l'autre. Les développeurs ayant réalisé des diagrammes de séquence comprenant des centaines d'objet comprendront très aisément l'intérêt de la chose.

Finalement, comme représenté dans le diagramme de séquence suivant, un objet peut être responsable tant de la création que de la disparition d'un autre. Un code C++ correspondant à ce diagramme de classe s'écrirait comme suit :

```
class O1 {
  public void jeConstruis(){
    O2 *unO2 = new O2() ;
  }

  public void jeDetruis(){
    delete unO2 ;
  }
}
```

Figure 10-17.

Un diagramme de séquence qui montre comment l’objet O1 peut avoir droit de vie et de mort sur l’objet O2, d’abord il le crée puis il le détruit.



L’instruction `delete` étant absente de Java et C#, un effet aussi net et efficace serait plus difficile à obtenir dans ces langages, et il faudrait passer par les bons et loyaux services du ramasse-miettes, en forçant son intervention par un appel explicite.

Le diagramme de séquence étant très proche du flot d’exécution d’un programme, on conçoit que l’évolution d’UML vers une forme de langage de programmation ait obligé à ajouter de nouveaux éléments graphiques qui rendent compte de la partie procédurale de ce flot. Cette évolution est très controversée, car elle alourdit considérablement ces mêmes diagrammes, ce qui rend l’utilisation de logiciels de développement UML inévitable. Certains développeurs, qui se sentent plus à l’aise avec une suite logique et écrite d’instructions procédurales, ne verront pas l’intérêt d’un tel enrichissement. La réalisation de ces diagrammes de séquence à la main et pour autant que l’on cherche à respecter fidèlement leur symbolique graphique (surtout l’emboîtement des cadres) tient du parcours du combattant. On comprend dès lors les réticences exprimées par les « UMListes du tableau noir », accrues davantage encore par la deuxième version d’UML. Si l’utilisation d’UML leur fait perdre du temps ou leur complique la vie, retour à l’expéditeur !! Il faudra revoir la copie. Vive les bons vieux langages de programmation. Il n’en reste pas moins que le souci d’universalisation d’UML au-delà de tous les langages de programmation continue à se vérifier, car tous ces langages reprennent ce type de mécanismes procéduraux (test, boucle,…) même si tous le font encore à leur guise et à partir d’une syntaxe légèrement modifiée. UML 2 permet, à nouveau, de transcender les différences, gommer la cosmétique, en se limitant à l’essentiel, les fonctionalités pures. Il permet d’intégrer dans cet esperanto OO, au départ uniquement dédié aux mécanismes OO, les mécanismes de la programmation procédurale. Plus rien ne veut ou ne peut lui échapper. L’entièreté de ce que vous avez fait en Python ou en Java, pourra faire l’objet d’une traduction simultanée en C++ ou en C#. Personnellement, je crois que cela vous permettrait une grosse économie de travail. On est preneur.

Exercices

Dans plusieurs des énoncés qui suivent, des relations d'héritage doivent être mises en œuvre. Nous vous conseillons donc de vous attaquer à ces énoncés-là après avoir lu le chapitre suivant.

Exercice 10.1

Tentez de dessiner les diagrammes de classe correspondant à la modélisation informatique des énoncés suivants :

1. Vous organisez un convoi humanitaire de véhicules. Les véhicules sont de trois sortes : camion, camionnette, voiture. Chacun des véhicules se caractérise par sa capacité tant à stocker des vivres qu'à transporter des passagers. Vous désirez prévoir à l'avance la consommation du convoi (dépendant, de manière différente pour chaque sorte de véhicule, de leur puissance et de leur charge).

2. Vous décidez de faire des travaux dans votre maison et vous vous interrogez quant aux dépenses à prévoir pour le paiement des salaires des ouvriers qui travailleront sur ce chantier. Vous savez que vous aurez affaire à plusieurs types d'ouvriers se différenciant par la façon dont ils veulent être payés. Certains sont déclarés, d'autres non. Certains veulent être payés à l'heure, d'autres par jour et d'autres encore par semaine. Finalement, certains veulent être payés cash, d'autres par chèque et d'autres encore par virement bancaire.

3. Vous réalisez un programme s'occupant de la gestion d'une petite « Cdthèque » de CD-Rom : éducatif, programme informatique et jeux, que vous souhaitez mettre à la disposition de vos amis pour une période de temps limité (maximum 10 jours). Le prix et la période maximale d'emprunt s'établissent différemment selon la nature des CD (on se base sur un échelon journalier pour les CD jeux, hebdomadaire pour les programmes et mensuel pour les CD éducatifs). Vos amis possèdent 4 types de matériel informatique : Mac, PC (avec Windows XP), PC (avec Windows 95), PC (avec Linux). Les CD-Rom et les informations qu'ils contiennent ne sont lisibles ou exécutables que sur certains de ces systèmes. Lorsqu'un de vos amis désire vous emprunter un ou plusieurs de vos CD, et ce pour une période désirée, votre programme doit être capable de :

   a. vérifier si ce CD est compatible avec son système informatique ;

   b. vérifier s'il est encore disponible ;

   c. lui indiquer combien cela lui coûtera ;

   d. expliquer à votre ami la procédure d'installation du CD, différente selon son système informatique (mais identique pour tous les CD) ;

   e. et de lui réclamer les CD qu'ils posséderaient encore et dont le temps d'emprunt est depuis dépassé.

4. Vous réalisez un programme s'occupant de la gestion d'un bureau de réservation pour spectacle. Votre programme vend des réservations pour une représentation (un certain jour à une certaine heure) d'un spectacle (caractérisé par son titre et son auteur). Un client, identifié par ses nom, adresse et numéro de téléphone, peut effectuer plusieurs réservations. Selon que le client est un abonné du bureau ou pas, il bénéficie d'une ristourne, d'une priorité sur les réservations et de facilités de paiement. Chaque réservation peut être de deux types, soit une réservation individuelle, soit une réservation de groupe. Dans les deux cas, des tickets sont délivrés au client : soit un ticket, soit autant de tickets que de personnes du groupe. À chaque ticket correspond une place pour la représentation.

5. Un organisme bancaire est propriétaire d'un certain nombre d'agences. Chaque agence possède un nombre important de clients et de comptes bancaires détenus par ces clients. Plusieurs clients peuvent avoir procuration sur un même compte et un même client peut posséder plusieurs comptes. L'organisme bancaire est également responsable d'un grand nombre de distributeurs que les clients peuvent utiliser pour tirer de l'argent ou consulter leurs comptes. À chaque compte sont associées une ou plusieurs cartes bancaires qui peuvent être de type différent, « carte bleue », « visa », « amex » et qui, selon leur type, permettent des modalités de crédit ou de remboursement qui peuvent varier. Seuls certains types de carte peuvent être utilisés dans un distributeur. Finalement, les comptes sont de deux sortes selon qu'ils peuvent être associés à une carte bancaire ou qu'ils ne le peuvent pas.

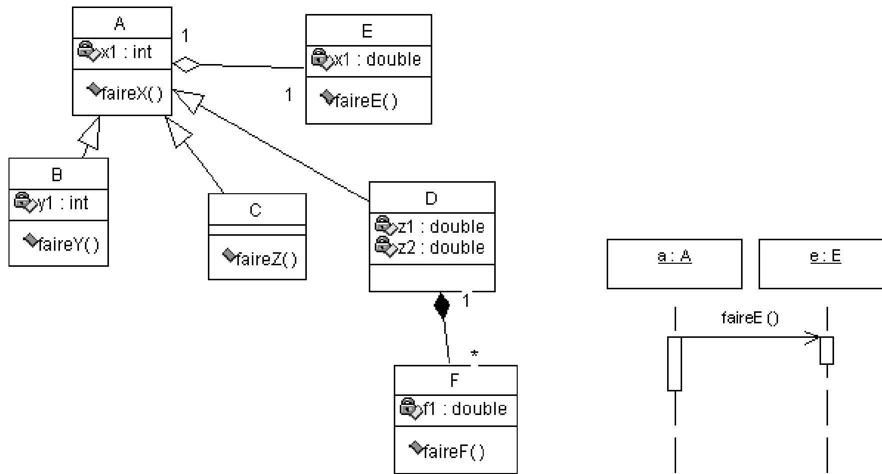
6. Vous devez réaliser la simulation d'un réseau ferroviaire sur lequel circulent différents types de train : des omnibus qui vont lentement et s'arrêtent à toutes les gares, des trains de marchandises qui vont moyennement vite et s'arrêtent dans une gare sur deux, et des trains à grande vitesse qui vont vite et ne s'arrêtent nulle part entre la gare de départ et la gare d'arrivée. Les trains de marchandises et à grande vitesse doivent ralentir à la vue de certains obstacles comme un feu de signalisation de couleur orange, un passage à niveau, un aiguillage ou une gare. Tous les trains doivent s'arrêter dès qu'un feu de signalisation passe au rouge, dès qu'un pylône est tombé sur la voie ou qu'un pont sur lequel le train doit passer est en pièces. Seuls les trains de marchandises et les omnibus s'arrêtent dans les gares. On simulera également le transport de la marchandise.

7. Vous réalisez la simulation d'un petit exercice de manœuvre militaire. Dans votre simulation, doivent apparaître les différents militaires avec leurs grades respectifs : général, colonel, capitaine, lieutenant et simple soldat. Tous les militaires sont capables de charger, de décharger leur arme, de déserter, etc., mais, selon leur grade, la manière de s'exécuter diffère. Chaque militaire doit s'en remettre à son responsable hiérarchique immédiat pour recevoir ses ordres de mission. Chaque militaire appartient à un régiment particulier. Les ordres de manœuvre sont donnés au régiment (par l'entremise du plus haut gradé) et ce par un QG central. Dans cette manœuvre, chaque régiment se voit désigner un emplacement initial et a comme objectif de conquérir un lieu particulier. En général, les ordres de manœuvre envoyés par le QG à chaque régiment peuvent être de trois types : « conquérir », « conquérir le lieu et y organiser un immense thé dansant », « conquérir le lieu et revenir chez soi avec un souvenir ». Chacun de ces types de manœuvre se particularise par une durée d'exécution, un budget à dépenser (et la manière de le dépenser), un nombre de militaires donné, ainsi qu'un ensemble de pratiques militaires particulières.

8. Vous devez réaliser la simulation d'un réseau ferroviaire sur lequel circulent différents types de train : des omnibus qui vont lentement et s'arrêtent à toutes les gares, des trains de marchandises qui vont moyennement vite et s'arrêtent dans une gare sur deux, et des trains à grande vitesse qui vont vite et ne s'arrêtent nulle part entre la gare de départ et la gare d'arrivée. Les trains de marchandises et à grande vitesse doivent ralentir à la vue de certains obstacles comme un feu de signalisation de couleur orange, un passage à niveau, un aiguillage ou une gare. Tous les trains doivent s'arrêter dès qu'un feu de signalisation passe au rouge, dès qu'un pylône est tombé sur la voie ou qu'un pont sur lequel le train doit passer est en pièces. Seuls les trains de marchandises et les omnibus s'arrêtent dans les gares. On simulera également le transport de la marchandise. Réalisez le diagramme de classe UML de cette application, en adaptant autant que faire se peut les principes de la programmation orientée objet.

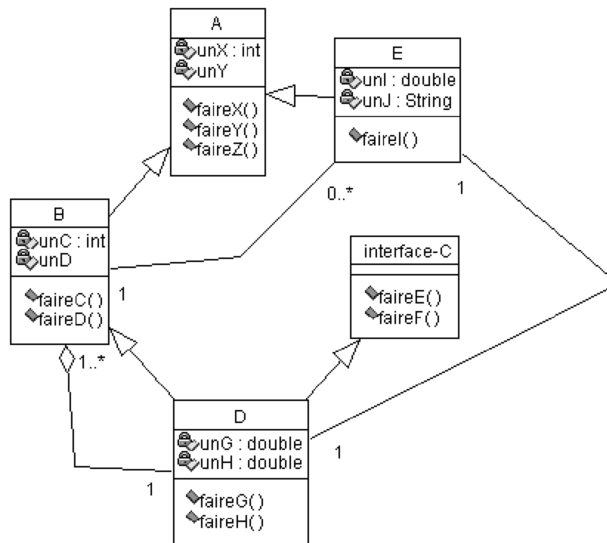
Exercice 10.2

Réalisez en C++ le squelette du programme qui accompagne ces diagrammes de classe et de séquence.



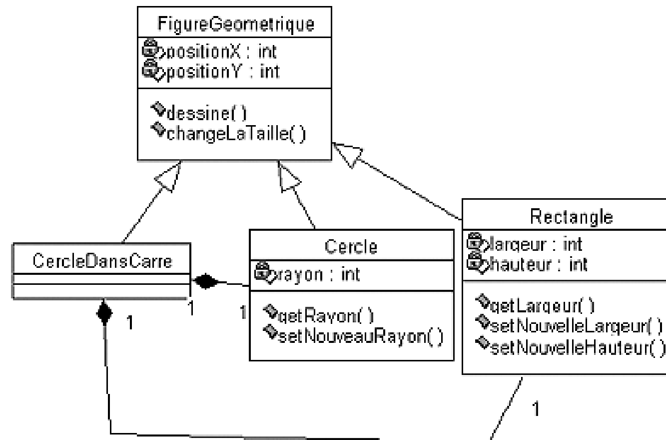
Exercice 10.3

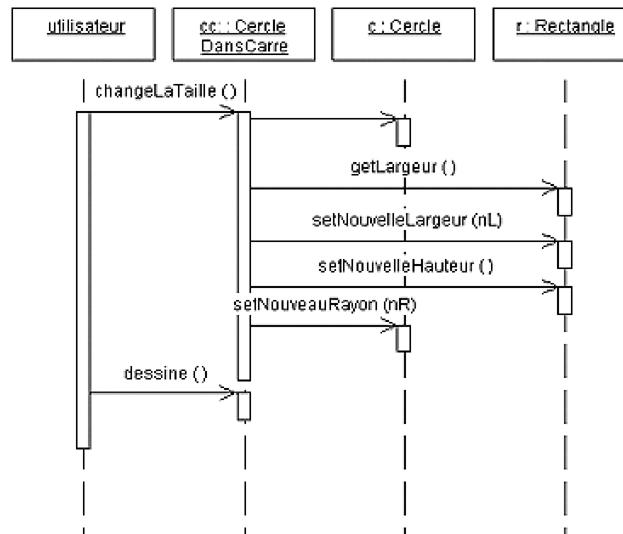
Réalisez le squelette de code Java que l'on pourrait générer automatiquement à partir du diagramme de classe suivant :



Exercice 10.4

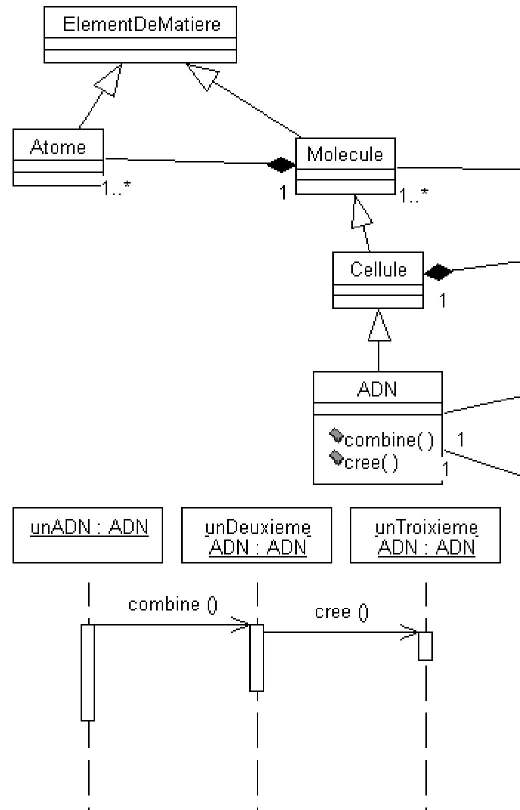
Réalisez le squelette de code C# que l'on pourrait générer automatiquement à partir de ces deux diagrammes UML :





Exercice 10.5

Réalisez le squelette de code C++ que l'on pourrait générer automatiquement à partir de ces deux diagrammes UML :



Exercice 10.6

Réalisez le diagramme de classe UML correspondant au code Java écrit ci-après (les deux colonnes se suivent :

```
import java.util.*;
class A {
  private B unB;
  public A() {}
  public void faireA() {}
}
class B extends C implements D {
  private Vector v = new Vector();
  public B() {
    for (int i=0; i<100; i++)
      v.addElement(new A());
  }
```

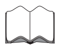
```
  public void faireD(E unE) {
    unE.faireE();
  }
}
class E {
  public void faireE() {}
}
interface D {
  public void faireD(E unE);
}
class C extends A {
  private A unA;
}
```

11

Héritage

Dans la poursuite de la modularisation mais verticale cette fois, ce chapitre introduit la pratique de l'héritage, comme, vers le haut, une factorisation dans la superclasse des attributs et des méthodes communs aux sous-classes, et, vers le bas, la spécialisation de ces sous-classes par l'addition des méthodes et des attributs qui leur sont propres.

Sommaire : Héritage simple — Principe de substitution — Héritage ou composition ? — Emploi de protected — Héritage multiple en C++ — Héritage public, private, protected, virtual



Doctus — Parmi plusieurs instances d'objets d'une même classe, l'ensemble des attributs permet de différencier les individus. Chacun d'eux pourra donc avoir une couleur ou une taille qui lui est propre.

Candidus — Mais que faire si nous voulons spécialiser un objet existant en lui ajoutant de nouveaux attributs ?

Doc. — Pour ça, l'OO nous propose le mécanisme d'héritage. Il nous permet de fabriquer ces objets à partir de ceux que nous avons sous la main tout en leur ajoutant ces nouveaux attributs.

Cand. — Intéressant… Ça donne un moyen économique pour fabriquer de nouvelles classes.

Doc. — C'est exact mais ils restent des représentants à part entière de toutes leurs superclasses : mêmes attributs, mêmes signatures de méthode. Même avec des attributs et méthodes supplémentaires, ils peuvent parfaitement ne jouer qu'un des rôles de base comme s'ils n'étaient pas spécialisés. Certaines superclasses peuvent juste servir d'intermédiaires pour réaliser plusieurs sous-classes.

Cand. — … et ces intermédiaires nous dispensent de dupliquer le tout dans chacune des classes concernées, c'est bien ça ?

Doc. — Exactement !

Cand. — Est-ce que nos sous-classes héritent vraiment tout de leurs parents et grands-parents ?

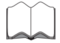
Doc. — Heureusement, non ! Que fais-tu donc du principe d'encapsulation ? Même pour le mécanisme d'héritage, les méthodes *private* ne concerneront que l'implémentation intime de chaque parent, les sous-classes n'y ayant pas accès.

Cand. — C'est vrai que, si quelqu'un se sert d'une de mes classes pour en faire hériter une des siennes, il ne serait pas avisé d'utiliser mes méthodes privées !

Doc. — En revanche, elles héritent des méthodes *protected*, c'est-à-dire les méthodes d'implémentation que tu veux mettre à disposition des sous-classes, tout en les rendant inaccessibles à toutes les autres.

Cand. — L'héritage est donc une porte ouverte qu'il peut être bon de refermer sur certains mécanismes intimes des parents.

Doc. — Une autre combinaison de classes, le multihéritage, constitue apparemment une économie par rapport à la composition. Il permet d'éviter les échanges de message nécessaires à la collaboration d'un composant. Toutefois, pour trouver une méthode de superclasse, cela devient plus complexe ; plusieurs chemins doivent être explorés,et il se peut même que nous devions choisir parmi plusieurs solutions possibles.

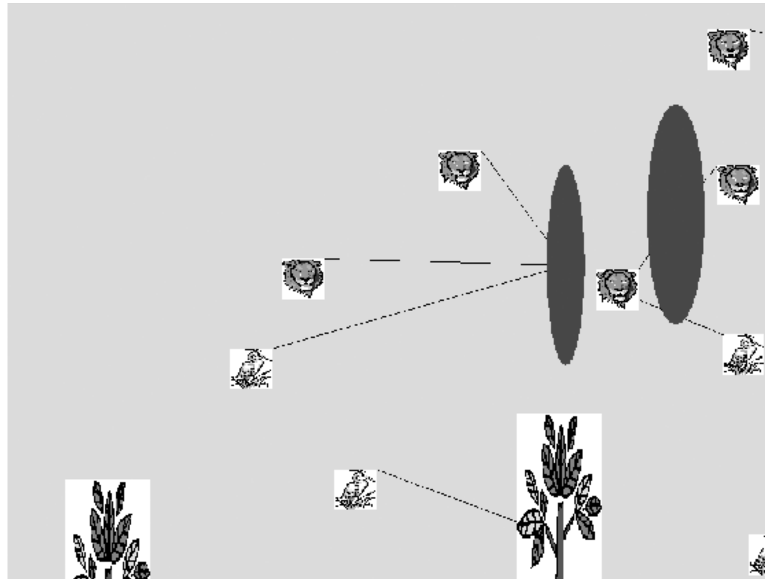


Comment regrouper les classes dans des superclasses

Reprenons l'exemple de notre petit écosystème du chapitre 3, dont une vue de la simulation est présentée ci- après. Le premier constat qui s'impose, c'est que nous avons démultiplié le nombre de proies, prédateurs, plantes et points d'eau. Nous n'allons pas nous priver de l'un des avantages premiers de la classe, qui est de donner naissance à une multitude d'objets sans se préoccuper, pour chacun, de re-préciser ce qu'il fait et de quoi il est fait.

Figure 11.1

Vue de la simulation finale du programme Java de l'écosystème.



Comme le diagramme de classe UML du chapitre précédent le montre parfaitement, jusqu'ici nous avons codé ce modèle à l'aide de 5 classes (oublions la vision pour l'instant). D'abord, les deux classes d'animaux : `Proie` et `Prédateur`, ensuite les deux classes de ressources naturelles : `Plante` et `Eau`, puis finalement la classe `Jungle`. Cette dernière agrège toutes les autres et lance la méthode principale qui, de manière itérée, fait évoluer les ressources et se déplacer les animaux. Vous aurez sans doute été sensible à ce petit dérapage sémantique, effectué sous contrôle bien sûr, et qui nous amène à réunir, sous le même concept d'animaux, la proie et le prédateur, ainsi que sous le même concept de ressource, l'eau et la plante.

Nous allons, en effet, joindre le geste logiciel à la parole, et introduire deux nouvelles classes `Faune` et `Ressource`, dont la raison d'être sera de regrouper ce qu'il y a de commun à la proie et au prédateur pour la première, et

de commun aux points d'eau et aux plantes pour la seconde. Voyons, dans un premier temps, les attributs communs à la proie et au prédateur, que nous installerons dorénavant dans la faune. Chacun se trouve situé en un point précis (x,y), chacun se déplace avec une vitesse projetée sur chaque axe (vitx, vity), chacun possède une énergie qui décroît suite aux efforts, et s'accroît grâce aux ressources, chacun est associé aux ressources disponibles dans la jungle, avec lesquelles ils interagissent.

Toutes ces propriétés se retrouveront dès à présent « plus haut dans les classes », c'est-à-dire dans la faune. Que reste-t-il, qui particularise et différencie encore la proie du prédateur ? Le prédateur interagit, en plus, avec les seules proies et vice versa. Les proies pouvant mourir, elles possèdent un attribut supplémentaire indiquant leur état de vie.

Toutes les plantes et tous les points d'eau sont caractérisés par les deux mêmes attributs : leur quantité et un compteur temporel qui sert à rythmer leur évolution naturelle (la plante qui pousse et l'eau qui s'évapore). Ici, la factorisation est encore plus radicale que dans le cas des animaux, car il ne reste, au bout du compte, aucun attribut qui différencie les plantes de l'eau.

Dans quelle mesure ne sont-ils pas alors simplement des objets différents d'une même classe `Ressource` ? Car si la seule différence qui subsiste entre les objets est la valeur de certains de leurs attributs, il n'y a plus lieu de découper les classes en sous-classes. Il n'existe pas de sous-classes de voiture rouge ou bleue, car ce sont simplement deux objets différents de la même classe voiture. Dans pareil cas, il suffit de s'en tenir, bien sûr, à la seule diversification des objets, qui sert justement à cela : encoder par chacun des valeurs d'attributs différentes. N'utilisez jamais l'héritage de manière abusive, pour ce qu'il n'est pas dans les langages OO, c'est-à-dire la différenciation des objets appartenant à une même classe par la seule valeur des attributs. Ne faites pas systématiquement de sous-classes pour les jeunes hommes et les hommes âgés, si seul leur âge les différencie, ou de sous-classes pour les voitures rapides ou lentes si, là encore, seule leur vitesse maximale les différencie et rien d'autre. Pour l'instant, nous nous sommes limités aux seuls attributs, et nous verrons bien vite que les méthodes jouent un rôle encore plus important, lors de cette factorisation dans une superclasse des caractéristiques communes aux sous-classes. À elles seules, elles justifieront, pour les ressources, la présence de ces deux niveaux hiérarchiques.

Héritage des attributs

Concentrons-nous, d'abord, sur l'héritage des attributs. Le diagramme UML ci-après (voir figure 11-2) illustre, à l'aide d'un nouveau symbole graphique, que nous avions délibérément laissé en suspens dans le chapitre précédent, le mécanisme d'héritage par les classes `Predateur` et `Proie` de la classe `Faune`, et par les classes `Plante` et `Eau` de la classe `Ressource`. Comme cela est visible sur le diagramme, de par la présence de la flèche d'héritage (seule la pointe la différencie de celle symbolisant le lien d'association dirigée), tous les attributs et les méthodes caractérisant la superclasse deviennent, automatiquement, attributs et méthodes de la sous-classe, sans qu'il n'y ait besoin de le préciser davantage. C'est la direction de la flèche qui spécifie lesquelles sont les superclasses et lesquelles sont les sous-classes, nullement leur position dans le diagramme de classe même s'il est de coutume d'installer les sous-classes en dessous des superclasses.

Première conséquence de cette application de l'héritage

Il ne peut y avoir dans la sous-classe, par rapport à sa superclasse, que des caractéristiques additionnelles ou des précisions. Ce que l'héritage permet d'abord, c'est de rajouter dans la sous-classe de nouveaux attributs et de nouvelles méthodes, les seuls à préciser dans la déclaration des sous-classes.

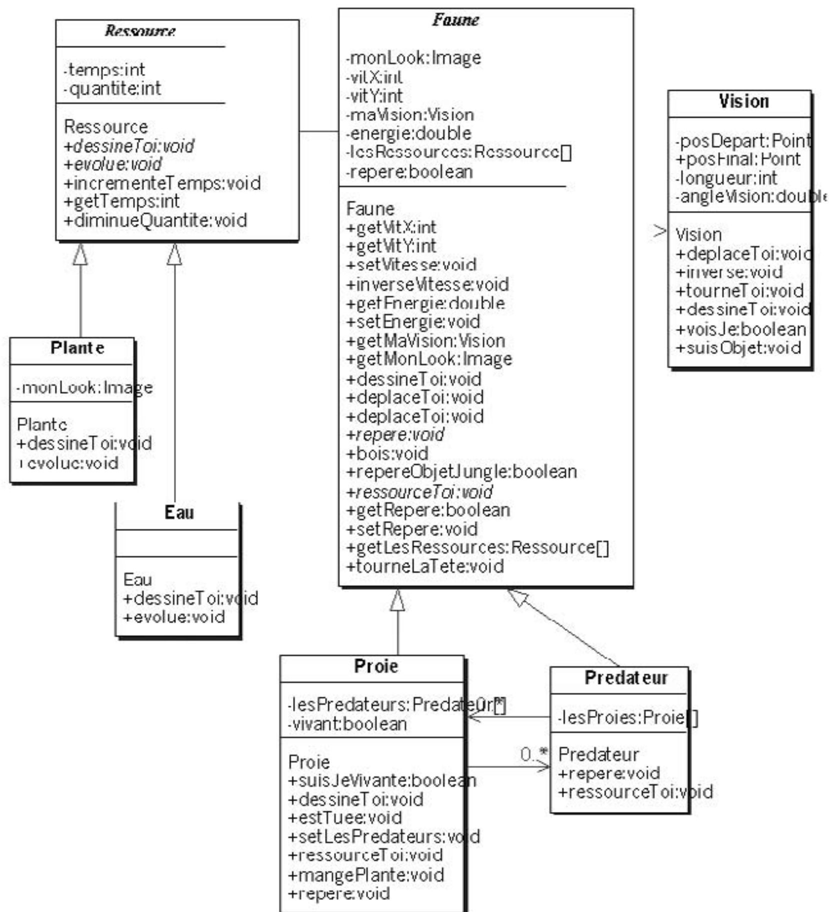


Figure 11-2

Diagramme de classe plus complet de l'écosystème où on voit apparaître deux superclasses : Faune et Ressource, et la manière dont les sous-classes héritent de celles-ci. Observez bien le sens et le dessin de la flèche symbolisant l'héritage. Les deux sont capitaux.

Ce que la relation d'héritage cherche à reproduire dans l'écriture logicielle, c'est l'existence, dans notre manière d'appréhender le monde, de concepts plus spécifiques et génériques. Nous le faisons tout naturellement pour des raisons d'économie déjà entrevue dans le premier chapitre, et nous retrouvons cette pratique « taxonomique » dans de nombreuses disciplines intellectuelles humaines : politique, économique, sociologique, biologique, zoologique… La possibilité de hiérarchiser notre conceptualisation du monde en différents niveaux d'abstraction nous permet une utilisation plus flexible de l'un ou l'autre de ces niveaux, selon le contexte d'utilisation. Alors que l'un de nous tape ce texte sur son portable posé sur la table de la salle à manger, l'informaticien du labo lui téléphone pour lui demander s'il souhaite une batterie de rechange pour un IBM ThinkPad. Sa compagne lui demande de retirer son ordinateur afin de pouvoir mettre la table et son enfant de cinq ans lui demande de pouvoir jouer sur sa machine. Quatre dénominations pour un même objet, quatre termes le désignant à différents niveaux d'abstraction, selon quatre contextes d'utilisation différents, quatre interlocuteurs et quatre besoins distincts. Chacun le désigne à sa manière, afin de communiquer son souhait le plus économiquement et le plus effectivement qui soit. Le choix du bon niveau d'abstraction se justifie par un souci d'optimisation de la communication, par le souhait « de dire le plus avec le moins ». Nul besoin de savoir qu'il s'agit d'un IBM ThinkPad pour réaliser que, tout IBM qu'il est, il encombre la table de la salle

à manger. Un concept est plus abstrait qu'un autre si, dans sa fonction descriptive, il englobe cet autre, s'il est plus général, plus passe-partout, plus adaptable. Comme la figure 11-3 l'illustre, il en est ainsi de « machine », plus abstrait que « ordinateur », plus abstrait que « portable », plus abstrait que « IBM ThinkPad ».

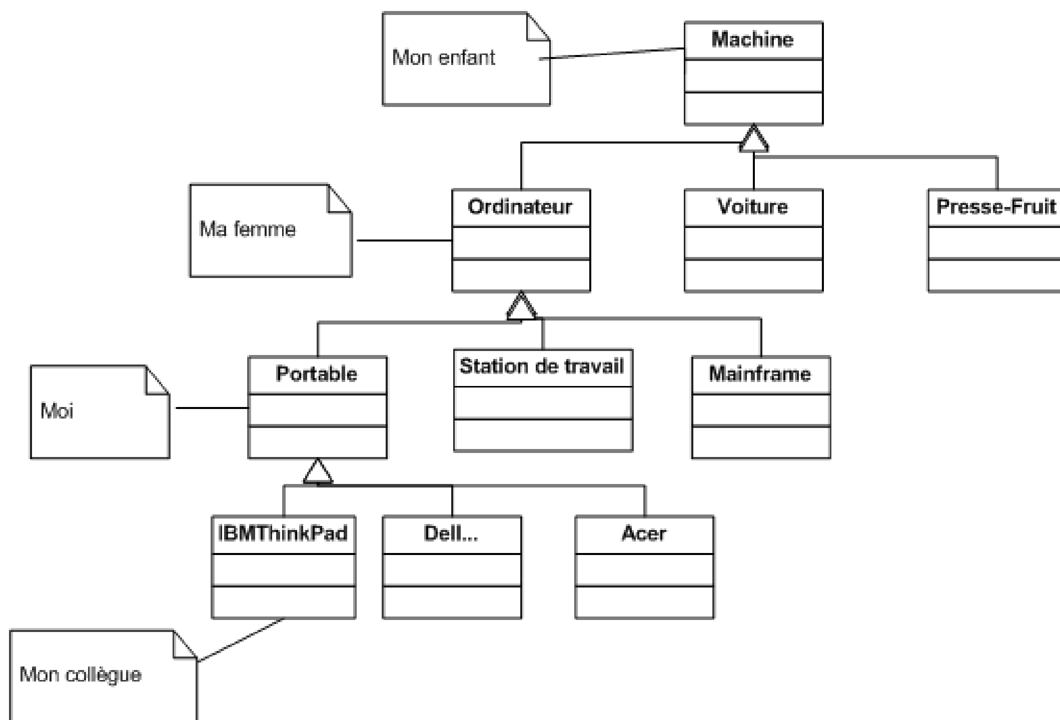


Figure 11-3

Le même ordinateur portable vu selon différents niveaux d'abstraction.

On vous montre une voiture et on vous demande de nous dire de quoi il s'agit. Il y a fort à parier que vous nous répondiez une « voiture » plutôt qu'une « Renault Kangoo » ou un « moyen de transport », ce qu'elle serait également. Un chien sera sans doute, un « chien », éventuellement un « caniche » mais certainement pas un « caniche nain ». Vous nous verrez taper sur un « portable » et non pas sur un « IBM ThinkPad ». Clairement, un des niveaux d'abstraction se voit privilégié par rapport aux autres. La raison en est simple, c'est le niveau le plus usité lors de toute communication d'information concernant, en effet, l'objet référé au cours de cette communication. C'est le niveau de base, celui qui caractérise le mieux l'objet, qui capture le plus d'informations sur les rôles et les fonctions qu'on lui attribue.

Pourquoi l'addition de propriétés ?

Pourquoi une classe plus spécifique ne peut-elle que rajouter des propriétés par rapport à sa superclasse ? Pour la bonne et simple raison qu'elle se doit de rester également cette superclasse, et qu'elle le restera, tant qu'elle partagera avec cette superclasse les mêmes propriétés. D'où la seule pratique valide qui consiste à trouver

dans la classe héritant un ajout d’informations par rapport à la classe héritée. Rien de ce qu’est ou de ce que fait une superclasse ne peut ne pas être ou ne pas être fait par toutes ses sous-classes. La sous-classe peut en faire plus, pour se démarquer, mais jamais moins. Ce mode de fonctionnement est évidemment transposable aux objets, instances des classes et sous-classes correspondantes, et donne lieu à un principe clé de la pratique orientée objet, principe qui est dit de « substitution ».

Principe de substitution : un héritier peut représenter la famille

Partout où un objet, instance d’une superclasse apparaît, on peut, sans que cela pose le moindre problème, lui substituer un objet quelconque, instance d’une sous-classe. Tout message compris par un objet d’une superclasse le sera obligatoirement par tous les objets issus des sous-classes. L’inverse est évidemment faux. Toute Ferrari peut se comporter comme une voiture, tout portable comme un ordinateur et tout livre d’informatique OO comme un livre. Vous pouvez le jeter par la fenêtre comme tout livre en effet. C’est pour cette simple raison qu’il sera toujours possible de typer statiquement un objet par une superclasse bien que, lors de sa création et de son utilisation, il sera plus précisément instance d’une sous-classe de celle-ci, par exemple « `SuperClasse unObjet = new SousClasse()` » (le compilateur ne bronche pas, même si l’objet typé superclasse sera finalement créé comme instance de la sous-classe ; c’est aussi la raison pour laquelle vous devez écrire deux fois le nom de la classe dans l’instruction de création d’objet) ou encore :

```
SuperClasse unObjet = new SuperClasse() ;
SousClasse unObjetSpecifique = new SousClasse() ;
unObjet = unObjetSpecifique ; // l’inverse serait refusé par le compilateur
```

Tout message autorisé par le compilateur lorsqu’il est censé s’exécuter sur un objet d’une superclasse peut tout autant s’exécuter sur un objet de toutes ses sous-classes. La Ferrari peut prendre des passagers ou simplement démarrer. L’inverse n’est pas vrai. Demandez à une voiture quelconque (une Mazda, par exemple) d’atteindre 300 km/h dans les 5 secondes et à un livre quelconque (la Bible, par exemple) de vous révéler les subtiles secrets de l’héritage en OO.

L’héritage : du cognitif aux taxonomies

Nous allons, au cours de notre développement, raffiner et illustrer ces différents fondements de la pratique de l’orienté objet. Mais, d’ici là, gardons à l’esprit que l’héritage a pour première vocation de reproduire ce mode

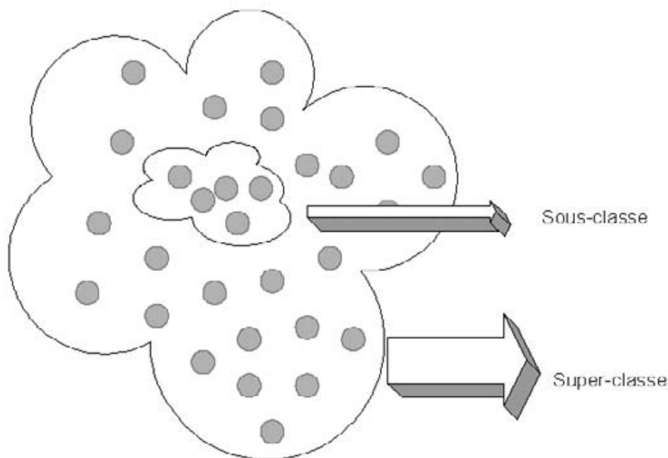


Figure 11-4
Interprétation ensembliste de l’héritage.

cognitif extrêmement puissant de conceptualisation hiérarchisée du monde, du plus général au plus spécifique. En conséquence, il n'y aura jamais lieu de le mettre en œuvre en OO autrement qu'entre des classes, qui, dans la conceptualisation que nous en avons, entrent dans ce rapport taxonomique. Si la source première d'inspiration de ce mécanisme puissant, permettant une organisation et un encodage économique de la connaissance, reste notre fonctionnement cognitif, il y a lieu dans une pratique, détachée maintenant des sciences cognitives, de tendre vers une formalisation plus rigoureuse et fiable.

L'intelligence artificielle, d'où est née en partie, par les travaux de Minsky, ce mécanisme informatisé d'héritage, a toujours, au départ d'une inspiration issue de notre fonctionnement cognitif, aspiré à une formalisation de ces mécanismes, pour les transposer dans une pratique d'ingénieur robuste et normative. Il en va ainsi de toutes les logiques classiques et moins classiques, des réseaux de neurones, de la théorie des probabilités subjectives, de la logique floue, autant d'outils ingénieristes qui trouvent leur origine dans les sciences cognitives.

Marvin Minsky

Marvin Minsky est un des pères fondateurs de l'intelligence artificielle. Il est depuis 50 ans une des grandes figures de cette discipline au célèbre MIT. En 1974, dans un article intitulé « A Framework for Representing Knowledge », il traite de la structuration de nos connaissances et de l'utilisation de celle-ci durant les processus cognitifs de résolution de problème, de compréhension du langage et de la perception visuelle. Dans cet article, il décrit des structures computationnelles particulières appelées « Frame », agrégat d'attributs, qui peuvent s'instancier lorsque les valeurs de ces attributs sont connues, et qui s'organisent dans notre cognition de manière hiérarchique, des plus génériques aux plus spécifiques. Ce fut une source d'inspiration certaine pour les premiers langages et développements orientés objet. Les auteurs dont Minsky s'inspira pour sa conception des Frames ne sont pas des informaticiens. Ses influences majeures sont les schémas du psychologue Jean Piaget, les paradigmes de l'épistémologue Thomas Khün ou les noumènes du philosophe Kant. Comme quoi, il faut de tout pour faire un bon et solide modèle informatique. Avant cela, pour avoir conçu les premiers réseaux de neurones et avoir clairement perçu leurs forces et leurs faiblesses, il fut, pendant 30 ans, le fossoyeur de l'approche neuronale en intelligence artificielle, redevenue très populaire depuis. On l'a décrit comme la sorcière du conte de Blanche-Neige, offrant la pomme empoisonnée à toute la communauté neuronale, très importante pendant les années 1950 et quasi léthargique jusqu'au début des années 1980.

Minsky est un fervent défenseur de l'approche symbolique en intelligence artificielle. Il ne nie pas le parallélisme inhérent à notre fonctionnement cérébral, mais il préfère appréhender celui-ci comme un ensemble d'agents spécialisés et travaillant de concert (comme autant d'objets s'envoyant des messages) pour la réalisation des tâches cognitives (cette vision a été popularisée dans son ouvrage *The Society of Mind*). Il déteste le tournant pris dans son propre laboratoire par des roboticiens outranciers, véritables apprentis sorciers, qui pensent qu'il suffit de mettre un robot embryonnaire, doté d'immenses facultés d'apprentissage, dans un environnement réel, pour le voir se transformer, avec le temps, en une créature intelligente. Il préfère l'approche moins empirique, qui consiste à davantage et mieux encore s'inspirer de nos processus mentaux, pour les reproduire le plus fidèlement possible dans des entités artificielles. Cependant, il ne pense pas que faire de la cognition humaine un système purement logique soit également la voie la plus prometteuse, vu l'immense flexibilité qui sous tend le raisonnement humain. En résumé, il pense que la seule voie prometteuse est la sienne et ses critiques dévastatrices à l'encontre de tout ce qui s'en éloigne sont devenues légendaires.

Ses intérêts sont multiples : les articulations mécaniques, l'éducation scolaire des jeunes enfants (avec un fidèle parmi les fidèles, Seymour Papert, ils ont conçu la petite tortue du « LOGO »), la possibilité d'extraterrestres et leurs éventuelles capacités intellectuelles, la musique, les mécanismes cognitifs sous-jacents à l'humour et aux blagues, les émotions (sujet de son tout dernier livre). Il participa à l'élaboration du scénario du chef-d'œuvre de Kubrick (tiré de l'œuvre d'Arthur C. Clarck), *2001 : l'Odyssée de l'espace*, surtout de la partie consacrée au célèbre et inquiétant HAL. Pour la petite histoire, le nom de cet ordinateur ne serait pas constitué des trois lettres qui précédent respectivement les lettres IBM, mais, plus simplement, viendrait de Heuristic Algorithm (en référence aux travaux de Minsky à l'époque). Tout était formidablement anticipé dans ce livre et film : la reconnaissance vocale, le jeu d'échecs automatisé, la lecture sur les lèvres, tout … Sauf qu'aucun des ordinateurs de bord n'affiche de fenêtre sur son écran et que nul souris n'est manipulée par les protagonistes.

Interprétation ensembliste de l'héritage

Ainsi, une possible interprétation, plus normative, de la pratique de l'héritage, passe tout simplement par la théorie des ensembles. Ce détour pourra vous être quelquefois utile, quand vous ressentirez un doute sur le pourquoi et le comment de la mise en œuvre de cette pratique dans un contexte donné. Comme cela est visible à la figure 11-3, la superclasse, comme « ensemble », regroupe en tant qu'instance tous les éléments, y compris tous ceux que regroupe l'ensemble sous-classe. C'est ce qui permet d'affirmer que tout objet d'une sous-classe est également objet de sa superclasse, et que l'on peut passer d'un type sous-classe à un type superclasse sans que cela ne cause de difficulté. L'inverse n'est évidemment plus vrai, car les sous-classes étant plus spécifiques, elles se permettent des choses qui ne sont pas du ressort des superclasses. Essayez en effet de faire rouler toutes les voitures comme des Ferrari, de faire aboyer tous les animaux comme des chiens, de pratiquer tous les sports comme vous pratiquez le ski, de trimballer tous les ordinateurs comme vous trimballez un portable, et vous serez vite convaincus de l'impossibilité de substituer à la sous-classe sa superclasse.

« Casting explicite » versus « casting implicite »

Comme nous le verrons dans le chapitre suivant, le « casting » permet à une variable typée d'une certaine façon lors de sa création d'adopter un autre type le temps d'une manœuvre. Conscient des risques que nous prenons en recourant à cette pratique, le compilateur l'accepte si nous recourons à un « casting explicite ». C'est le cas en programmation classique quand on veut, par exemple, assigner une variable réelle à une variable entière. Dans le cas du principe de substitution, les informaticiens parlent souvent d'un « casting implicite », signifiant par là, qu'il n'y a pas lieu de contrer l'opposition du compilateur quand nous faisons passer un objet d'une sous-classe pour un objet d'une superclasse. Le compilateur (gardien du temple de la bonne syntaxe et de la bonne pratique) ne bronchera pas, car cette démarche est tout à fait naturelle et acceptée d'office. En revanche, faire passer un objet d'une superclasse pour celui d'une sous-classe requiert la présence d'un « casting explicite », par lequel vous prenez l'entière responsabilité de ce comportement risqué. Le compilateur vous met en garde (vous êtes en effet sur le point de commettre une bêtise) mais, ensuite, vous en faites ce que vous voulez en votre âme et conscience. Par exemple, si vous transférez un réel dans un entier (la superclasse dans la sous-classe), c'est en effet une bêtise, car vous perdez la partie décimale. En général, vous en êtes pleinement conscients et assumez la responsabilité de cette perte d'information. Les puristes de l'informatique détestent la pratique du casting (explicite évidemment) qui est toujous évitable par un typage plus fin et une écriture de code plus soignée. Un mauvais casting sera responsable d'une erreur à l'exécution du code lorsque l'on assigne à une classe un type qui ne lui correspond pas, et ce alors que le compilateur a laissé faire.

Qui peut le plus peut le moins

Revenons aux ensembles. Cette manière de concevoir de l'héritage (la plus solide) a permis de contrer un collègue qui affirmait qu'en se basant sur le seul rajout d'attributs, il était possible de voir la classe des nombres réels comme une sous-classe de celle des nombres entiers, ou les rectangles comme une sous-classe des carrés. En effet, un réel peut être simplement vu comme un entier auquel on ajouterait l'attribut valeur décimale et un rectangle comme un carré auquel on ajoute un côté supplémentaire. Or, il n'est point besoin de pouvoir suivre la démonstration du théorème de Fermat pour savoir que les nombres entiers sont un sous-ensemble des nombres réels et les carrés un sous-ensemble des rectangles, et qu'en prolongeant cette vision, la bonne, les entiers et les carrés deviennent respectivement, non plus la superclasse, mais la sous-classe des réels et des rectangles. Et c'est ce qu'ils sont en effet, en informatique tout comme en mathématique. En vérité, la notion de classe et de sous-classe se base essentiellement sur la nature opératoire des objets qui en découle. Tout ce que vous faites avec un réel en informatique, vous pouvez le faire avec un entier. L'inverse est faux. Par exemple, quand vous dimensionnez une fenêtre sur un écran, les programmes qui le font s'attendent à recevoir un entier, et ne titillez pas le compilateur en transmettant un réel à la place. En revanche, la situation inverse laissera le compilateur aussi froid que les circuits qui le font fonctionner.

Pour se sortir du dilemme des attributs, il serait correct de dire que, tous comme les réels, les entiers ont : l'attribut entier plus l'attribut décimal, mais ils sont beaucoup plus spécifiques que les réels, en ceci que la valeur de leur attribut décimal est forcément nulle. On pourrait croire que, au vu de cette spécificité accrue, la sous-classe en fera toujours plus que la superclasse. Cependant, ce qui fait la spécificité de la sous-classe est, dans le même temps, ce qui risque d'être le moins sollicité par les autres classes. Il n'est pas rare que la sous-classe passe plus de temps à jouer le rôle de sa superclasse qu'à se laisser aller à exprimer vraiment ce qu'elle a au plus profond des tripes, comme nous le verrons dans la suite. Finalement, à observer les diagrammes de classe UML, vous constaterez que les rectangles des sous-classes sont très souvent plus petits que ceux des superclasses, car ils ne se différencient que par quelques ajouts.

Héritage ou composition ?

Un objet de type sous-classe est d'autant plus un objet de type superclasse que son stockage en mémoire se compose, d'abord, d'un objet de type superclasse, puis d'un espace mémoire réservé aux attributs qui lui sont propres, comme indiqué dans la figure ci-après. Ce mode de stockage ressemble à s'y méprendre au mode de stockage d'un objet, qui serait en partie composé d'un autre objet en son sein. Cela revient à dire qu'eu égard au stockage des objets en mémoire, rien ne différencie vraiment un lien de composition entre deux classes d'un lien d'héritage entre ces deux même classes. Nous verrons qu'il n'en va plus de même lors de l'appel des méthodes.

Dans le cas de la composition, il y a bien envoi de messages de la classe qui offre le logement à celle qui est logée. Dans le cas de l'héritage, on ne parlera plus d'envoi de message, car il s'agit bien d'une méthode propre à la classe elle-même, quitte à être héritée d'une autre. N'oublions pas que l'héritage crée une vraie fusion entre les deux classes, alors que la composition maintient un rapport de clientélisme. Certains programmeurs tendent à favoriser tant que faire se peut la composition au détriment de l'héritage. Nous défendons ici la position classiquement admise : faites parler les concepts de la réalité que vous cherchez à reproduire et écoutez-les. C'est la réalité que vous cherchez à dépeindre qui doit avoir le dernier mot. Si deux entités qui vous intéressent entrent dans une relation taxonomique (comme la Ferrarri et la voiture), recourez à l'héritage, dans

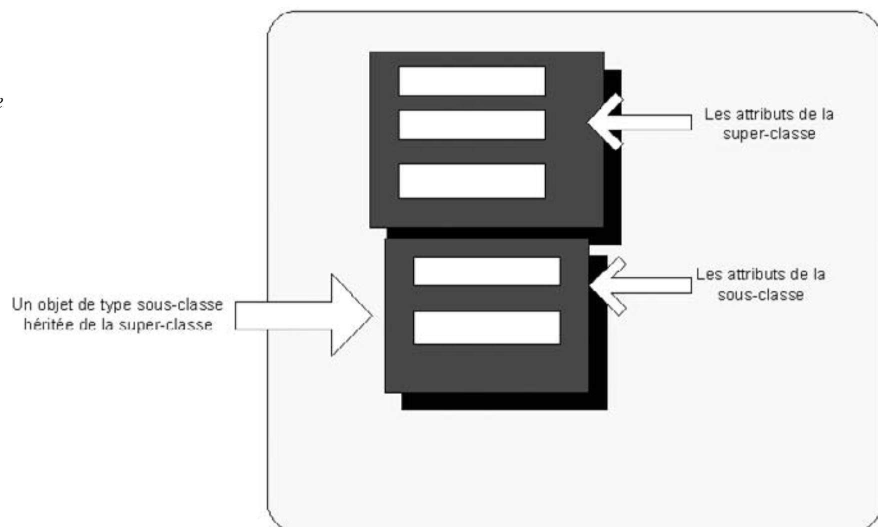


Figure 11-5

Un objet de la sous-classe est stocké en mémoire, d'abord comme un objet de la superclasse, puis en rajoutant les attributs qui lui sont propres.

tous les autres cas (comme le moteur et la voiture) choisissez la composition, sans oublier bien sûr les autres possibilités que sont l’agrégation ou l’association (comme la voiture et son propriétaire).

Juste pour en rajouter une petite couche, et vous convaincre que si le monde était simple, on ne posséderait sans doute plus les facultés nécessaires à le conceptualiser, la réalité elle-même, du moins ce que l’on en conçoit, n’est pas toujours aussi tranchée entre la composition et l’héritage. Vous êtes une espèce d’animal, ne le prenez pas mal, car vous aurez beaucoup de mal à contenir l’animal qui est en vous, n’est-ce pas ? Le jour où vous vous demanderez s’il y a quoi que ce soit à lire dans ce livre que nous nous efforçons d’écrire, au moins vous nous aurez fait le plaisir d’illustrer, une nouvelle fois, l’ambiguïté qu’il peut y avoir entre composition et héritage.

Économiser en rajoutant des classes ?

La pratique d’héritage en OO a été introduite pour favoriser l’économie de représentation et une simplification accrue. Or, dans le seul exemple vu jusqu’à présent, le programme a été plus alourdi par l’addition de deux nouvelles classes qu’autre chose. En matière d’économie, c’est assez discutable. Sept classes, c’est plus que cinq, et, par ailleurs, cette démarche de factorisation n’est pas forcément des plus élémentaires.

Il est clair que cet effort ne portera ses fruits que si le programme s’enrichit de l’addition de nouvelles ressources et de nouveaux animaux. Nous pourrions rajouter une multitude d’autres espèces de proies et de prédateurs, différentes des précédentes. Nous pourrions penser également à un ensemble de nouvelles ressources. Après un certain nombre d’additions, nous aurons largement rentabilisé ce préalable effort de factorisation, par une épargne en écriture et l’évitement de possibles erreurs, causées par l’addition de codes redondants. Dans notre cognition également, l’utilisation de ces super-concepts se renforce avec la multiplication des concepts qui en sont dérivés.

L’héritage favorise la réutilisation de code existant, de surcroît s’il a été écrit par un informaticien plus aguerri que vous, ce qui mâche considérablement la besogne, en permettant de ne vous concentrer que sur votre seul apport. Héritez d’une des classes `collections` déjà pré-codées en Java pour y encoder vos voitures, vos petits amis et petites amies, les examens de fin d’année… et vous héritez gratuitement d’une série de fonctionnalités bien utiles, comme, la possibilité d’ordonner très simplement les éléments de cette collection (algorithme qui aura été écrit par un pro). Java recourt largement à l’héritage, afin que vous puissiez récupérer dans l’écriture de vos classes un ensemble de librairies pré-codées, utiles à la réalisation d’interface graphique, de gestion d’événements, de programmation concurrentielle, etc.

La place de l’héritage

L’héritage trouve parfaitement sa place dans une démarche logicielle dont le souci principal devient la clarté d’écriture, la réutilisation de l’existant, la fidélité au réel, et une maintenance de code qui n’est pas mise à mal par des évolutions continues, dues notamment à l’addition de nouvelles classes.

Héritage des méthodes

Passons maintenant aux méthodes. À l’instar de l’héritage des attributs, les méthodes s’héritent également, et toute sous-classe peut ajouter de nouvelles méthodes lors de sa déclaration. Comme schématisé par le diagramme UML de la figure 11-6, lorsque la classe `O1` désire communiquer avec la classe `FilleO2`, elle a la possibilité, soit de lui envoyer les messages qui sont propres à cette classe, soit de lui envoyer tous ceux hérités de la classe `O2`. Dans la famille classe, je demande la fille… Quand la proie ou le prédateur consomme l’eau ou

la plante, elles envoient à l'eau ou la plante le même message `diminueQuantite()`. Ce message n'est ni déclaré dans la classe `Eau` ni dans la classe `Plante`, mais elles en héritent toutes deux de leur superclasse `Ressource`. Pour les proies et les prédateurs aussi, de nouvelles méthodes communes aux deux, comme `tournerLaTete()`, `repereUnObjet()`, peuvent être déclarées plus haut dans la classe `Faune`. Cela permet à toute classe nécessitant d'interagir avec les proies ou les prédateurs de le faire directement avec la faune « qu'il y a en eux », sans se préoccuper de savoir exactement de quelle faune il s'agit.

> **Messages et niveaux d'abstraction**
>
> L'héritage permet à une classe, communiquant avec une série d'autres classes, de leur parler à différents niveaux d'abstraction, sans qu'il soit toujours besoin de connaître leur nature ultime (on retrouvera ce point essentiel dans l'explication du polymorphisme), ce qui facilite l'écriture, la gestion et la stabilisation du code.

Vous dites que vous avez eu un accident de voiture. Cela suffit pour vous plaindre, sans avoir besoin de connaître la marque de la voiture, sauf si vous êtes garagiste ou assureur. Aussi, dans le diagramme UML de l'écosystème de la figure 11-6, on s'aperçoit que la classe `Faune` interagit avec la classe `Ressource` car, quelle que soit la ressource, l'objet `Faune` pourra envoyer à l'objet `ressource` un message, qui ne nécessitera pas de connaître la nature de la ressource en question.

Voici également un petit bout de code extrait de la déclaration de la classe Faune

```
public boolean repereObjetJungle(ObjetJungle unObjet)
{
  repere = false;
  if (maVision.voisJe(unObjet.getMaZone()))
  {
    vitX = (int)((unObjet.getMaZone().x
          + (unObjet.getMaZone().width/2)
          - getPosX()) * energie);
    vitY = (int)((unObjet.getMaZone().y
          + (unObjet.getMaZone().height/2)
          - getPosY()) * energie);
    maVision.suisObjet(unObjet.getMaZone());

    repere = true;
  }
  return repere;
}
```

Le seul point d'intérêt que présente ce bout de code est l'apparition d'une nouvelle classe qui, dans un premier temps, a été omise, pour alléger le diagramme de classe, et passée comme argument de la méthode `repereObjetJungle ()`. Il s'agit de la super-superclasse `ObjetJungle`. Cette méthode, apparaissant dans la classe `Faune`, a pour fonction unique de vérifier si la vision dont la faune est composée rencontre un objet quelconque de la jungle : une autre faune ou une ressource, et, si c'est le cas, de forcer l'objet `vision` à suivre cet objet repéré. Ici, ce repérage concerne indifféremment n'importe quel objet de la jungle. Bien évidemment, la suite des événements dépendra de la nature intime de l'objet : proie, prédateur, plante ou eau. Mais la fonctionnalité repérage, elle, peut se désintéresser de cette nature intime. On s'aperçoit, dès lors, de l'intérêt qu'il y a à rajouter une nouvelle superclasse au-dessus de `Faune` et `Ressource`, comme illustré ci-après. Ce dernier

diagramme constitue de fait la dernière et définitive version de notre programme. La classe `ObjetJungle` ne contient que les coordonnées de la position de tous les objets, qu'ils se déplacent ou pas (récupérable par la méthode `getMaZone()`). La seule partie des faunes et des ressources qui compte pour la vision, c'est leur position, c'est-à-dire l'`ObjetJungle` dont ils sont constitués. Ces seules coordonnées sont tout ce qui importe à la vision pour repérer un objet. Autrement dit, la vision ne nécessite d'interagir qu'avec la partie `ObjetJungle` de tous les objets de la jungle.

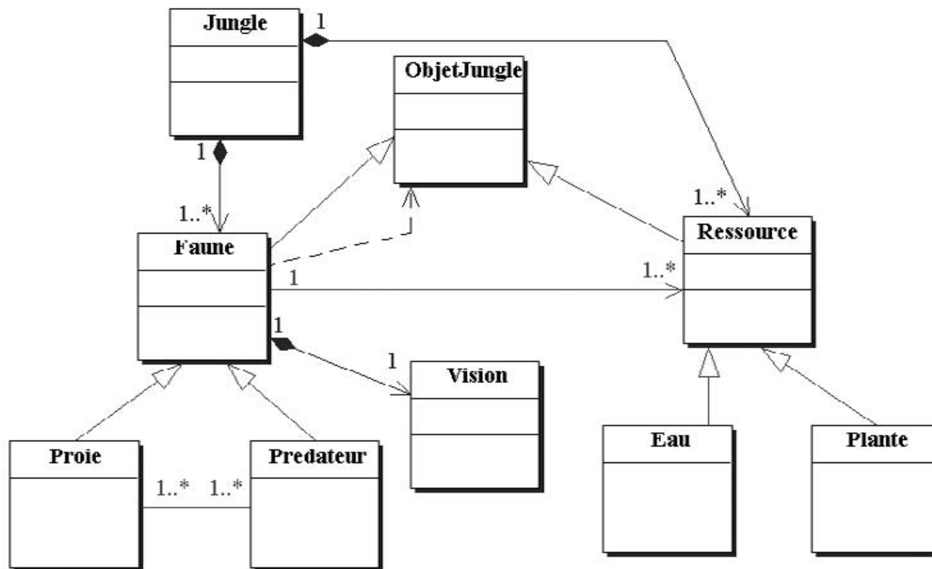


Figure 11-6
Diagramme de classe complet de l'écosystème.

Dans le petit diagramme UML qui suit, figure 11-7, et les codes qui le traduisent respectivement dans les trois langages, la classe `O1` peut s'adresser à la classe `FilleO2` en tant que `FilleO2` ou en tant qu'`O2`. Elle a, en définitive, la possibilité d'envoyer deux messages à la classe `FilleO2 : jeTravaillePourO2()` ou `jeTravaillePourLaFilleO2()`. En plus, la classe `O1`, dans une autre de ses méthodes, reçoit un argument de type « `FilleO2` ». Cela nous permet d'illustrer le principe de substitution, qui dit que l'on pourra appeler cette méthode, en lui passant indifféremment un argument de type « `FilleO2` » ou un argument de type « `O2` ».

Code Java

```
class O1 {
  private FilleO2 lienFilleO2;
  public O1(FilleO2 lienFilleO2) {
    this.lienFilleO2 = lienFilleO2;
  }
  public void jeTravaillePourO1() {
    lienFilleO2.jeTravaillePourO2();
    lienFilleO2.jeTravaillePourLaFilleO2();
```

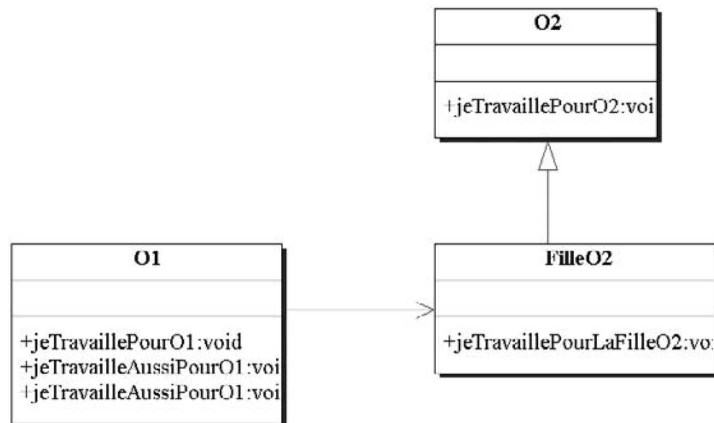


Figure 11-7

La classe O1 communique avec la classe FilleO2 en lui envoyant des messages qui sont, soit hérités de O2, soit propres à la classe héritière.

```
    }
    /* dans les résultats montrés, cette méthode est
       d'abord active, puis mise en commentaire */
    public void jeTravailleAussiPourO1(FilleO2 lienFilleO2) {
      lienFilleO2.jeTravaillePourO2();
      lienFilleO2.jeTravaillePourLaFilleO2();
    }
    public void jeTravailleAussiPourO1(O2 lienO2) {
      lienO2.jeTravaillePourO2();
    }
  }
  class O2 {
    public O2() {}
    public void jeTravaillePourO2() {
      System.out.println("Je suis un service rendu par la classe O2");
    }
  }
  class FilleO2 extends O2 { /* C'est la syntaxe de l'héritage en java */
    public FilleO2() {}
    public void jeTravaillePourLaFilleO2() {
      System.out.println("Je suis un service rendu par la classe FilleO2");
    }
  }
  public class Heritage1 {
    public static void main(String[] args) {
      O2 unObjetO2 = new O2();
      FilleO2 uneFilleO2 = new FilleO2();
      O1 unObjetO1 = new O1(uneFilleO2);
      unObjetO1.jeTravaillePourO1();
      unObjetO1.jeTravailleAussiPourO1(unObjetO2);
      unObjetO1.jeTravailleAussiPourO1(uneFilleO2);
    }
  }
```

Résultats

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
```

Résultats avec la méthode jeTravailleAussiPourO1(FilleO2 lienFilleO2) mise hors d'action

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O2
```

La différence essentielle dans ce code Java réside dans le fait que la première méthode `jeTravailleAussiPourO1()`, codée pour recevoir un argument de type superclasse `O2`, sera maintenant appelée avec un argument de type sous-classe, sans que cela ne pose le moindre problème (au vu du casting implicite présenté plus haut). Remarquez également que la surcharge d'une méthode par le simple fait qu'un des arguments soit de la sous-classe d'un argument de la méthode surchargée ne pose aucune difficulté car, à l'exécution, il est facile de choisir la bonne méthode à exécuter selon le type statique de l'argument que l'on passe dans la méthode. S'il s'agit d'un argument de type superclasse, il ne peut s'agir que de la méthode prévue à cet effet. En revanche, s'il s'agit d'un argument de type sous-classe et si elle existe, on choisira d'exécuter la méthode prévue à cet effet (voir le prochain chapitre). Sinon on peut se rabattre sur la méthode censée être exécutée sur un objet de type superclasse et dont la sous-classe hérite de toute manière.

Code C#

```
using System;
class O1 {

  private FilleO2 lienFilleO2;
  public O1(FilleO2 lienFilleO2) {
    this.lienFilleO2 = lienFilleO2;
  }
  public void jeTravaillePourO1() {
    lienFilleO2.jeTravaillePourO2();
    lienFilleO2.jeTravaillePourLaFilleO2();
  }
  public void jeTravailleAussiPourO1(FilleO2 lienFilleO2) {
    lienFilleO2.jeTravaillePourO2();
    lienFilleO2.jeTravaillePourLaFilleO2();
  }
  public void jeTravailleAussiPourO1(O2 lienO2) {
    lienO2.jeTravaillePourO2();
  }
}
```

```
class O2 {
  public O2() {}
  public void jeTravaillePourO2() {
    Console.WriteLine("Je suis un service rendu par la classe O2");
  }
}
class FilleO2 : O2  { /* C'est la syntaxe de l'héritage en C# plus proche du C++ */
  public FilleO2() {}
  public void jeTravaillePourLaFilleO2() {
    Console.WriteLine("Je suis un service rendu par la classe FilleO2");
  }
}
public class Heritage1 {
  public static void Main() {
    O2 unObjetO2 = new O2();
    FilleO2 uneFilleO2 = new FilleO2();
    O1 unObjetO1 = new O1(uneFilleO2);
    unObjetO1.jeTravaillePourO1();
    unObjetO1.jeTravailleAussiPourO1(unObjetO2);
    unObjetO1.jeTravailleAussiPourO1(uneFilleO2);
  }
}
```

Résultats

```
… les mêmes qu'en Java.
```

Code C++

```
#include "stdafx.h"
#include "iostream.h"
class O2 {
  public:

    O2() {}
    void jeTravaillePourO2() {
      cout << "Je suis un service rendu par la classe O2" << endl;
    }
};
class FilleO2 : public O2 { /* C'est la syntaxe de l'héritage en C++, notez la présence du
➥" public " que nous justifierons dans la suite */
  public:
    FilleO2() {}
    void jeTravaillePourLaFilleO2() {
      cout << "Je suis un service rendu par la classe FilleO2" << endl;
    }
};
```

```
class O1 {
  private:
    Fille02* lienFille02;
  public:
    O1(Fille02* lienFille02) {
      this->lienFille02     = lienFille02;
    }
    void jeTravaillePourO1() {
      lienFille02->jeTravaillePourO2();
      lienFille02->jeTravaillePourLaFille02();
    }
    void jeTravailleAussiPourO1(Fille02* lienFille02) {
      lienFille02->jeTravaillePourO2();
      lienFille02->jeTravaillePourLaFille02();
    }
    void jeTravailleAussiPourO1(O2* lienO2) {
      lienO2->jeTravaillePourO2();
    }
    void jeTravailleAussiPourO1(Fille02 lienFille02) {
      lienFille02.jeTravaillePourO2();
      lienFille02.jeTravaillePourLaFille02();
    }
    void jeTravailleAussiPourO1(O2 lienO2) {
      lienO2.jeTravaillePourO2();
    }
};
int main(int argc, char* argv[]) {
  O2* unObjetO2Tas          = new O2();
  Fille02* uneFille02Tas    = new Fille02();
  O1* unObjetO1             = new O1(uneFille02Tas);
  unObjetO1->jeTravaillePourO1();
  unObjetO1->jeTravailleAussiPourO1(unObjetO2Tas);
  unObjetO1->jeTravailleAussiPourO1(uneFille02Tas);
  cout <<endl << "Essais avec des objets piles" <<endl<< endl;
  O2 unObjetO2Pile;
  Fille02 uneFille02Pile;

  O1* unAutreObjetO1        = new O1(&uneFille02Pile);
  unAutreObjetO1->jeTravaillePourO1();
  unAutreObjetO1->jeTravailleAussiPourO1(&unObjetO2Pile);
  unAutreObjetO1->jeTravailleAussiPourO1(&uneFille02Pile);
  cout <<endl << "Derniers essais avec des objets piles" <<endl<< endl;
  unAutreObjetO1->jeTravaillePourO1();
  unAutreObjetO1->jeTravailleAussiPourO1(unObjetO2Pile);
  unAutreObjetO1->jeTravailleAussiPourO1(uneFille02Pile);
  return 0;
}
```

Résultats

Rien de vraiment spécial ne se produit dans ces différents essais, testant l'héritage avec des objets stockés sur la pile ou sur le tas. Le même résultat est obtenu trois fois. En C#, rien de semblable ne peut être produit si l'on choisit de recourir à des objets dont le temps de vie est géré par la mémoire pile, car les « structures » qui le permettent ne peuvent simplement pas hériter entre elles. En conséquence et malgré l'existence des structures en C#, le seul recours pour des jeux d'héritage comme ceux-ci est, tout comme en Java, de se limiter aux seuls référents et à la mémoire tas.

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
Je suis un service rendu par la classe O2 (écrit deux fois)
Je suis un service rendu par la classe FilleO2
```

Essais avec des objets piles

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
Je suis un service rendu par la classe O2 (écrit deux fois)
Je suis un service rendu par la classe FilleO2
```

Derniers essais avec des objets piles

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe FilleO2
Je suis un service rendu par la classe O2 (écrit deux fois)
Je suis un service rendu par la classe FilleO2
```

Code Python

```
class O1:
        lienFilleO2=None

      def __init__(self,lienFille2):
            self.__lienFilleO2=lienFille2
      def jeTravaillePourO1(self):
            self.__lienFilleO2.jeTravaillePourO2()
            self.__lienFilleO2.jeTravaillePourLaFilleO2()
      def jeTravailleAussiPourO1(self,lienO2):
            if isinstance(lienO2,FilleO2):
                 lienO2.jeTravaillePourO2()
                 lienO2.jeTravaillePourLaFilleO2()
            else:
                 lienO2.jeTravaillePourO2()

class O2:
      def __init__(self):
            pass
      def jeTravaillePourO2(self):
                 print "je suis un service rendu par la classe O2"
```

```
class Fille02(02): #remarquez la manière dont Python réalise l'héritage
      def __init__(self):
            pass
      def jeTravaillePourLaFille02(self):
                 print "Je suis un service rendu par la classe Fille02"

unObjet02=02()
uneFille02=Fille02()
unObjet01=01(uneFille02)
unObjet01.jeTravaillePour01()
unObjet01.jeTravailleAussiPour01(unObjet02)
unObjet01.jeTravailleAussiPour01(uneFille02)
```

Comme Python ne type pas les paramètres des méthodes et que la surcharge de méthode est impossible, il est difficile de restituer le même exemple. Cependant, quelque chose de très approchant est présenté dans la version Python. Cela suffit à illustrer le fait que les méthodes peuvent être héritées et qu'en fonction du type dynamique de l'objet (ici testé par le biais de l'instruction `isinstance`), la méthode choisie sera celle de la superclasse ou de la sous-classe.

Code PHP 5

```
<html>
<head>
<title> Héritage et substitution </title>
</head>
<body>
<h1> Héritage et substitution </h1>
<br>

<?php
   class 01 {
      private $lienFille02;

      public function __construct($lienFille02) {
           $this->lienFille02 = $lienFille02;
      }

      public function jeTravaillePour01() {
           $this->lienFille02->jeTravaillePour02();
           $this->lienFille02->jeTravaillePourLaFille02();
      }

      public function jeTravailleAussiPour01($lien02){
           if ($lien02 instanceof Fille02) {
                  $lien02->jeTravaillePour02();
                  $lien02->jeTravaillePourLaFille02();
           } else {
                  $lien02->jeTravaillePour02();
           }
     }
}
```

```
   class O2 {
      public function __construct() {}

      public function jeTravaillePourO2() {
           print("je suis un service rendu par la classe O2 <br> \n");
      }
}

   class FilleO2 extends O2 { //heritage PHP = syntaxe Java
      public function __construct() {}

      public function jeTravaillePourLaFilleO2() {
           print("je suis un service rendu par la classe FilleO2 <br> \n");
      }
}

   $unObjetO2 = new O2();
   $uneFilleO2 = new FilleO2();
   $unObjetO1 = new O1($uneFilleO2);
   $unObjetO1->jeTravaillePourO1();
   $unObjetO1->jeTravailleAussiPourO1($unObjetO2);
   $unObjetO1->jeTravailleAussiPourO1($uneFilleO2);

   ?>

</body>
</html>
```

Dans le code PHP 5, on retrouve une syntaxe de l'héritage proche de Java, par l'entremise du mot-clé `extends` et tout comme en Python, en l'absence de typage, l'utilisation de l'expression `instanceof()` aussi empruntée à Java, permet de vérifier quelle version de la méthode doit vraiment s'exécuter.

La recherche des méthodes dans la hiérarchie

Les méthodes des superclasses et des sous-classes étant, comme les attributs, stockées ensemble, mais dans la mémoire des méthodes cette fois, lors de l'envoi du message d'`O1` vers la `FilleO2`, la méthode recherchée le sera, d'abord, dans la zone mémoire correspondante au type de l'objet, c'est-à-dire la zone mémoire `FilleO2`. Si la méthode ne s'y trouve pas, grâce à l'instruction d'héritage (comme indiqué à la figure 11-8), on sait que cette méthode peut se trouver plus haut, quelque part dans une superclasse. La montée en cordée, de superclasse en superclasse, à la découverte de la méthode recherchée, peut-être longue, et tout dépendra de la profondeur de la structure hiérarchique d'héritage réalisée dans l'application logicielle. Toutes les méthodes dans la hiérarchie peuvent s'appliquer sur l'objet, car le compilateur aura bien vérifié que chacune, quel que soit le niveau hiérarchique où elle se trouve, n'interférera qu'avec les attributs et les méthodes qui existent à ce niveau.

Ces montées et descentes, pendant l'exécution du programme, à la recherche de la méthode appropriée à exécuter sur l'objet, ont amené certains à parler d'un fonctionnement de type « yoyo ». Sans conteste, les voyages dans la RAM ralentissent considérablement toute l'exécution d'un programme. Or, on a déjà rencontré ces déplacements en examinant l'activation successive d'objets, qui peuvent se trouver stockés n'importe où dans la RAM. On accroît ce phénomène, en le reproduisant du côté des méthodes, dont la quête peut également occasionner ces périples incessants.

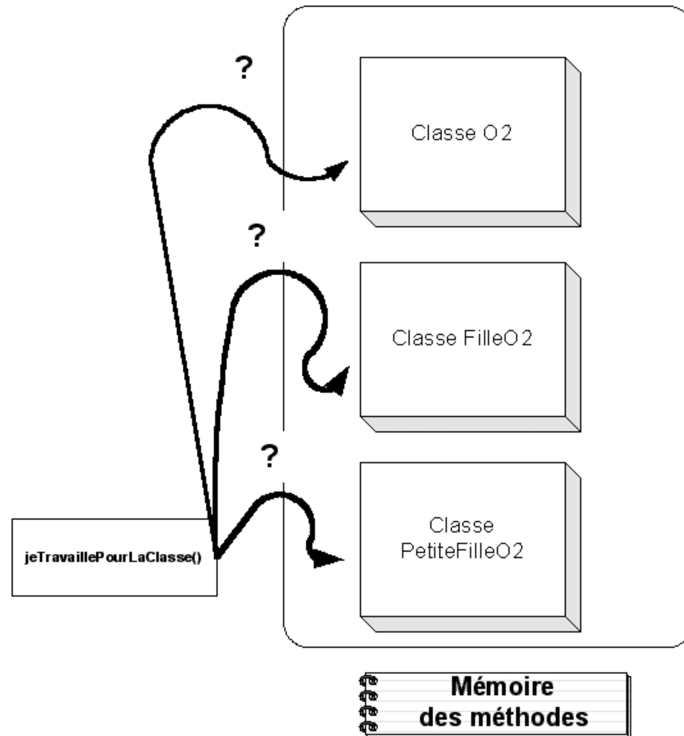


Figure 11-8
Recherche de la méthode de superclasse en superclasse dans la structure hiérarchique de classes.

Rien de bien original à répondre à cette critique fondée si ce n'est, une nouvelle fois, d'accepter la programmation OO pour ce qu'elle est : une approche plus simplifiée, plus intuitive, plus stable, et dont la complexification est mieux maîtrisée, du développement logiciel, et non pour ce qu'elle n'est pas, une volonté d'exploitation à tous crins des possibilités d'optimisation liée au fonctionnement intime des processeurs. Il s'agit bien d'un parti pris OO contre processeur.

Encapsulation protected

`protected` est une troisième manière, en plus de `private` et `public`, de caractériser tant les attributs que les méthodes d'une classe. Il s'agit de raffiner le souci d'encapsulation, discuté aux chapitres 7 et 8, en l'adaptant à la pratique d'héritage. Rappelons les deux raisons premières de cette pratique d'encapsulation, qui consiste à tenter de maximiser la partie privée des classes au détriment de leur partie publique.

Concernant les attributs et les seules valeurs qui sont tolérées à leur égard, il faut laisser à leur classe, et à aucune autre, le soin de gérer leur intégrité. Ensuite, pour les attributs comme pour les méthodes, il est logique d'anticiper de possibles changements dans le codage d'une classe, et souhaitable de minimiser au mieux l'impact de ces changements sur les classes qui interagissent avec elle. L'addition de `protected` vient d'un souci légitime, ayant pour objet le statut des sous-classes par rapport aux autres classes.

Dans la société, pour qui a l'esprit de famille, il est indéniable que l'héritier est un être privilégié dans une famille. Mais dans la programation OO, les sous-classes doivent-elles être privilégiées ou logées à la même

enseigne que toutes les autres classes ? Question légitime car, en effet, un attribut ou une méthode `protected` rendra son accès possible, en plus de la classe où ils se trouvent déclarés, aux seules sous-classes héritant de celle-ci. Quoi de plus légitime que de permettre aux héritiers d'hériter de leur dû le plus facilement qui soit ? Si l'héritier ne jouit d'aucun privilège par rapport à la première classe venue, quelle espèce d'héritage est-ce donc là ?

Protected

Un attribut ou une méthode déclaré `protected` dans une classe devient accessible dans toutes les sous-classes. La charte de la bonne programmation OO déconseille l'utilisation de « protected ». D'ailleurs, Python ne possède pas ce niveau d'encapsulation.

Si ce souci de statut est compréhensible, d'où, de fait, la possibilité qui reste offerte aux programmeurs d'utiliser l'accès `protected`, l'avidité des héritiers est à ce point déconnectée du motif premier de l'encapsulation que la charte du bon programmeur OO bannit l'utilisation de `protected`. Python ne permet d'ailleurs pas ce niveau d'encapsulation. En effet, même par rapport à toutes ses sous-classes, la superclasse se doit de préserver son intégrité. De même, tout changement dans une partie de code `protected` affectera toutes les sous-classes. Pour toutes les classes, séparées dans leur écriture logicielle d'une classe concernée, il vaut mieux renforcer la sécurité et la stabilité, en ne laissant publique qu'une faible partie du code de la classe, publique pour toutes les autres classes, quelle que soit leur proximité sémantique avec la classe concernée.

Les héritières resteront toujours privilégiées, en ceci que, malgré un accès plus indirect, via les méthodes, elles posséderont les mêmes attributs que la superclasse, et en ceci, aussi, qu'elles pourront, à loisir, ré-utiliser les méthodes de cette dernière, sans recourir à l'envoi de messages. Ce sont leurs méthodes à elles aussi. Elles pourront les appeler à l'intérieur de leur code, comme s'il s'agissait de méthodes définies par elles. Néanmoins, dans l'organisation logicielle et les tracas causés par sa répartition entre une équipe de programmeurs, rien ne contribue vraiment à distinguer une sous-classe d'une autre.

Malgré les réserves exprimées à l'égard de l'encapsulation `protected`, une utilisation très courante de l'encapsulation `protected`, par exemple dans les librairies Java, se retrouve dans la définition de méthodes de superclasse que l'on encourage l'utilisateur de ces superclasses à redéfinir dans les sous-classes (nous préciserons cela dans le prochain chapitre). Il faut les redéfinir en bas en faisant explicitement appel à ces méthodes de là-haut (d'où le `protected` pour n'autoriser cet appel que par les sous-classes). `protected` incite alors à une redéfinition, en faisant toutefois appel aux méthodes originelles de la superclasse de départ. Nous clarifierons cela dans les prochains chapitres.

Héritage et constructeurs

Comme nous l'avons vu précédemment, il est fréquent que la sous-classe ajoute des attributs par rapport à la superclasse. Tout objet, instance de la sous-classe, possède dès lors deux ensembles d'attributs, ceux qui lui sont propres et ceux hérités de là-haut. Se pose alors le problème de la pratique des constructeurs, que nous savons être indispensable, en tous cas vivement conseillée, pour l'initialisation de ces attributs lors de la création de chaque objet. Comment doit se comporter le constructeur de la sous-classe dans le traitement des attributs qui ne lui incombent qu'indirectement, c'est-à-dire par héritage ? Java, C# et C++ se comportent de la même façon, que nous allons décortiquer grâce à trois petits codes Java, qui visent à clarifier cet aspect assez subtil de la programmation objet. Python et PHP 5, que nous verrons à la fin, se particularisent.

Premier code Java

```
class O1 {
       protected int unAttributO1; // attribut protected

       public O1() {
           this.unAttributO1 = 5; // le constructeur initialise l’attribut
       }
}

class FilsO1 extends O1 {
       private int unAttributFilsO1;

       public FilsO1() {} /* ici, le constructeur de la superclasse est appelé par défaut ou
                          de manière implicite */

       public void donneAttribut() {
           System.out.println("mes Attributs sont: " + unAttributO1 + " " + unAttributFilsO1);
           /* l’attribut de O1 est accessible grâce au « protected »
       }
}

public class TestConsHerit {
       public static void main(String[] args) {
           FilsO1 unFils = new FilsO1();
           unFils.donneAttribut();
       }
}
```

Résultats

```
mes Attributs sont 5 0
```

Ce code Java est élémentaire sauf sur un point. Une classe `O1` possède un attribut que nous déclarons `protected` pour pouvoir y accéder dans les sous-classes. Le constructeur de cette classe initialise l’attribut à 5. Une classe `FilsO1` est déclarée qui hérite de `O1` et possède un attribut supplémentaire. Apparamment, le constructeur de la sous-classe ne fait rien. Or, et toute la subtilité est là, si nous découvrons le résultat du code, nous constatons que le constructeur de la superclasse a pourtant été appelé car l’attribut hérité de la superclasse vaut bien 5. Nous voyons à l’œuvre un mécanisme implicite, commun à Java, C# et C++ et absent des langages de script comme Python et PHP 5 : un constructeur de la superclasse sans argument est toujours appelé par défaut par la sous-classe. Soit il a été défini, comme dans ce code-ci, soit Java en propose un par défaut, qui se limite à initialiser tous les attributs à des valeurs par défaut : `0` ou `null`. S’il est défini, il se substitue purement et simplement à celui par défaut.

Deuxième code Java

```
class O1 {
       protected int unAttributO1; // attribut protected

       public O1(int unAttributO1) {
           this.unAttributO1 = unAttributO1;
       }
```

```
}

class Fils01 extends 01 {
       private int unAttributFils01;

       public Fils01(int unAttribut01, int unAttributFils01) {
           this.unAttributFils01 = unAttributFils01;
       }

       public void donneAttribut() {
           System.out.println("mes Attributs sont: " + unAttribut01 + " " + unAttributFils01);
       }
}

public class TestConsHerit {
       public static void main(String[] args) {
           Fils01 unFils = new Fils01(5,10);
           unFils.donneAttribut();
       }
}
```

Ce code est assez logique dans sa forme, le constructeur de la superclasse s'occupe d'initialiser son attribut et celui de la sous-classe le sien. Pourtant, le compilateur fait des siennes et gromelle qu'il ne trouve plus aucun constructeur ne recevant aucun argument, et pour cause : le nouveau constructeur de la superclasse `01` a balayé celui-ci afin de le remplacer par un constructeur à un argument, la valeur initiale de l'attribut. Si l'idée de laisser chaque constructeur s'occuper de ses propres attributs est plutôt une bonne idée, il reste à forcer la sous-classe à appeler le constructeur de la superclasse avec la valeur initiale de l'attribut qui le concerne, comme le code Java ci-dessous, qui compile et s'exécute sans problème, l'illustre.

Troisième code Java : le plus logique et le bon

```
class 01 {
       protected int unAttribut01; // attribut protected

       public 01(int unAttribut01) {
           this.unAttribut01 = unAttribut01;
       }
}

class Fils01 extends 01 {
       private int unAttributFils01;

       public Fils01(int unAttribut01, int unAttributFils01) {
           super(unAttribut01) /* appel explicite du constructeur, doit être la 1ère instruction */
           this.unAttributFils01 = unAttributFils01;
       }

              public void donneAttribut() {
           System.out.println("mes Attributs sont: " + unAttribut01 + " " + unAttributFils01);
       }
}
```

```
public class TestConsHerit {
       public static void main(String[] args) {
           Fils01 unFils = new Fils01(5,10);
           unFils.donneAttribut();
       }
}
```

Résultats

```
mes Attributs sont : 5 10
```

Le constructeur de la sous-classe fait appel (et il doit le faire en premier), par l'entremise de `super()`, au constructeur de la superclasse. Ici `super` est simplement un pointeur vers la superclasse, pointeur que nous retrouverons pas plus tard que dans le prochain chapitre. Ici, l'instruction `super()` se borne à rappeler le constructeur de la superclasse. Pourquoi, de fait, faire appel au constructeur de la superclasse ? Simplement, dixit le compilateur, parce qu'on n'a pas le choix. Chaque classe s'occupe de l'initialisation de ses propres attributs. Gardez toujours à l'esprit le découpage fort des responsabilités en OO. Rendez à chaque classe ce qui appartient à chaque classe. Chacun à sa classe... et les attributs seront bien gardés.

Héritage et constructeur

La sous-classe confiera au constructeur de la superclasse (qu'elle appellera par l'entremise de `super()` en Java et `base` en C# et `parent` en PHP 5) le soin d'initialiser les attributs qu'elle hérite de celle-ci. C'est une excellente pratique de programmation OO que de confier explicitement au constructeur de la superclasse le soin de l'initialisation des attributs de cette dernière. D'ailleurs, si vous ne le faites pas, Java, C# et C++ le font par défaut, en appelant implicitement un constructeur sans argument.

Nous ajoutons ici les versions C# et C++, parfaitement équivalentes, à quelques détails de syntaxe près, au troisième petit code Java ci-dessus.

En C#

```
using System;

class O1 {
       protected int unAttributO1;

       public O1(int unAttributO1) {
           this.unAttributO1 = unAttributO1;
       }
}

class FilsO1:O1 {
       private int unAttributFilsO1;

       public FilsO1(int unAttributO1, int unAttributFilsO1):base(unAttributO1) {
           /* notez la version différente de l'appel au constructeur de la superclasse */
           this.unAttributFilsO1 = unAttributFilsO1;
       }
```

```
        public void donneAttribut() {
                  Console.WriteLine("mes Attributs sont: " + unAttribut01 + " " + unAttributFils01);
        }
}

public class TestConsHerit {
          public static void Main() {
                   Fils01 unFils = new Fils01(5,10);
                   unFils.donneAttribut();
        }
}
```

La seule vraie différence est l'appel au constructeur de la superclasse qui se fait par le mot-clé `base` plutôt que `super`, et dès la déclaration de la méthode plutôt que dans le corps d'instructions. Cela garantit qu'il s'agira en effet de la 1ère instruction éxécutée.

En C++

```
#include "stdafx.h"
#include "iostream.h"

class O1 {
protected:
       int unAttribut01;

public:
       O1(int unAttribut01) {
           this->unAttribut01 = unAttribut01;
       }
};

class Fils01:public O1 {
private:
       int unAttributFils01;

public:
       Fils01(int unAttribut01, int unAttributFils01):O1(unAttribut01) { /* appel du constructeur
                                   de la superclasse */
           this->unAttributFils01 = unAttributFils01;
       }

       void donneAttribut() {
           cout << "mes Attributs sont: " <<unAttribut01<<" "<<unAttributFils01<<endl;
         }
};

int main()
{
       Fils01* unFils = new Fils01(5,10);
       unFils->donneAttribut();
       return 0;
}
```

La syntaxe de l’appel du constructeur de la superclasse est très proche du C# (dans la déclaration plutôt que dans le corps d’instructions), à ceci près qu’il faut explicitement faire référence au nom de la superclasse. Comme nous le verrons par la suite, le C++ permet le multihéritage, ce qui rend les mots-clés `super` et `base` parfaitement ambigus.

En Python

```
class O1:
       unAttributO1=0

       def __init__(self,unAttributO1):
              self.unAttributO1 = unAttributO1;

class FilsO1(O1):
       unAttributFilsO1=0

       def __init__(self,unAttributO1,unAttributFilsO1):
           O1.__init__(self,unAttributO1) #appel du constructeur de la superclasse
           self.unAttributFilsO1 = unAttributFilsO1

       def  donneAttribut(self):
              print "mes Attributs sont: %s" % self.unAttributO1,self.unAttributFilsO1

unFils = FilsO1(5,10)
unFils.donneAttribut();
```

À la différence des trois autres langages, Python ne fait jamais d’appel implicite au constructeur de la superclasse. Dès lors, tout appel doit s’expliciter. Si ce n’est pas le cas, le code s’exécute malgré tout, mais les attributs de la superclasse ne seront pas initialisés. Par économie d’écriture, et le `protected` n’existant pas dans Python, les attributs ont été laissés publics. Finalement, il faut, comme pour le C++ avec lequel Python partage l’acceptation du multihéritage, indiquer le nom de la superclasse dont on déclenche le constructeur.

En PHP 5

```
<html>
<head>
<title> Héritage des constructeurs </title>
</head>
<body>
<h1> Héritage des constructeurs </h1>
<br>

<?php
   class O1 {
       protected $unAttributO1;

       public function __construct($unAttributO1) {
              $this->unAttributO1 = $unAttributO1;
       }
   }

   class FilsO1 extends O1 {
       private $unAttributFilsO1;
       /* Il faut obligatoirement appeler le constructeur de la
       superclasse */
```

```
        public function __construct($unAttribut01, $unAttributFils01) {
            parent::__construct($unAttribut01); // attention à la syntaxe avec « parent »
            $this->unAttributFils01 = $unAttributFils01;
        }

        public function donneAttribut() {
            print("mes attributs sont: $this->unAttribut01 et $this->unAttributFils01 <br> \n");
        }
    }

    $unFils = new Fils01(5,10);
    $unFils->donneAttribut();

?>

</body>
</html>
```

Comme en Python, l'appel du constructeur de la superclasse est obligatoire pour initialiser les attributs de la superclasse. Comme PHP 5 n'admet que l'héritage simple, tout comme Java et C#, la référence ci-dessus se fait cette fois par l'utilisation du mot-clé `parent`.

Héritage public en C++

C++ n'est pas avare de subtilités et de mécanismes sophistiqués. D'aucuns les décrieront comme tordus et inutiles, alors que d'autres les qualifieront, émerveillés, de méga-puissants et de vitaux. Parmi ceux-ci, un héritage, au lieu d'être `public` (comme vous pouvez le constater dans le code C++ plus haut), peut alternativement être déclaré comme `private` ou `protected`. Comme indiqué dans le diagramme de classe qui suit, la différence entre ces trois héritages réside dans l'accès des attributs et méthodes de la superclasse, lorsqu'une classe associée à la sous-classe désire accéder à ceux-ci.

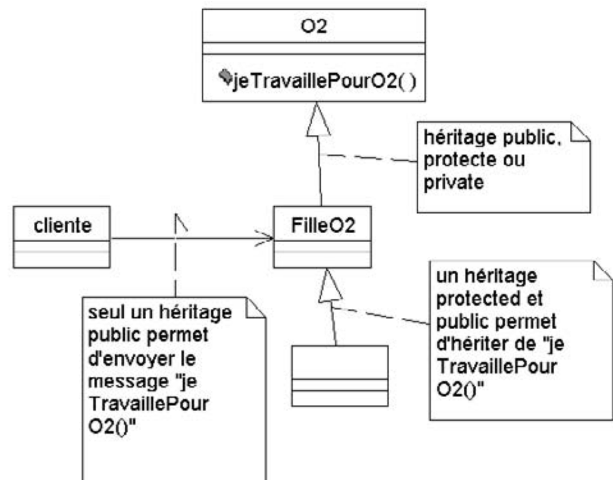


Figure 11-9

Différence en C++ entre les héritages public, protected et private.

Limitons-nous aux seules méthodes publiques dans la superclasse. Si l'héritage est `public`, ce qui est très majoritairement le cas, les méthodes publiques héritées de la superclasse deviennent également publiques pour toutes les classes. Nous avons pris l'héritage public comme le fonctionnement par défaut, quand la classe `O1` pouvait envoyer à la `fille02` des messages dont le corps se trouvait déclaré, soit directement dans la `fille02`, soit hérité de `02`. De fait, des héritages autres que `public` ne sont pas possibles dans les autres langages. Lors d'un héritage privé, une méthode `public` devient `private` dans la sous-classe, et la même deviendra `protected` dans la version `protected` de l'héritage. Ceux que ce mécanisme séduit le justifieront par un renforcement encore plus marqué de l'encapsulation, car il devient possible de limiter davantage encore l'impact de modifications dans les parties publiques. Mais comme les méthodes `public` le sont par ailleurs pour les empêcher de trop changer, cette sévérité accrue apparaît quelque peu exagérée.

Nous retrouverons souvent dans C++ une offre bien plus abondante de degrés de liberté à sélectionner ou calibrer. En C++, tout ce qui pouvait être imaginé comme trucs et ficelles de programmation l'a été. C'est au programmeur, alors, de procéder pour chacun de ces degrés au bon calibrage. Ce calibrage requérant une compréhension suffisante des conséquences de chacun des choix, compréhension faisant défaut chez de nombreux programmeurs, les langages OO plus jeunes ont fait le choix de se débarrasser d'un grand nombre de ces degrés de liberté (un choix qui semble plutôt leur réussir). Comme nous avons déjà eu l'occasion de le dire et le redire, C++ est à Java ce que sont les supers appareils photo 24 × 36, super réflex et autres aux simples instamatics. Beaucoup de règlages en plus, mais qui n'empêchent pas les instamatics de faire souvent de bien meilleures photos. Trop de règlages nuit. Trop de liberté tue la liberté (cela aurait pu être de Sartre, mais c'est de nous).

Le multihéritage

Il n'y a rien de conceptuellement dérangeant à ce qu'une classe puisse hériter de plusieurs superclasses à la fois. Un artiste de cirque est souvent un clown, un trapéziste, un musicien, un jongleur et un dompteur, tout à la fois. Un ordinateur portable est en même temps un ordinateur et un bagage. Il hérite des deux fonctionnalités : on l'allume, le « boote », le « back-up » (désolé pour le français, sorry vraiment…) mais, également, on le passe dans le détecteur de métaux, on l'enlève de sa malette devant les membres du service de sécurité de l'aéroport avant d'enlever ses chaussures, on le glisse dans le compartiment à bagages ou sur le siège arrière d'une voiture. Il a donc deux chances de se faire voler : soit en tant qu'ordinateur, soit en tant que bagage. Un livre d'informatique est un livre et aussi l'objet de mille rancœurs et frustrations pour beaucoup d'étudiants plongés dans ce livre.

Ramifications descendantes et ascendantes

Notre conceptualisation du monde s'arrange bien de cette multiplicité qui, de manière plus formelle, élargit la structure de l'héritage : d'arbre (quand l'héritage ne peut se ramifier que de manière descendante), en graphe (quand les ramifications peuvent se faire autant dans le sens descendant – plusieurs sous-classes pour une classe – qu'ascendant – plusieurs superclasses pour une classe, voir la figure qui suit). En principe, toute classe pourrait réunir en son sein des caractéristiques différentes provenant de plusieurs superclasses. Il suffit qu'elle les additionne.

Or, les langages Java, C# et PHP 5 interdisent le multihéritage (en partie, ils l'autorisent pour les interfaces comme nous le verrons plus bas), alors que C++ et Python l'autorisent totalement. Tout le problème provient de la nécessité pour les caractéristiques héritées d'être vraiment différentes entre elles.

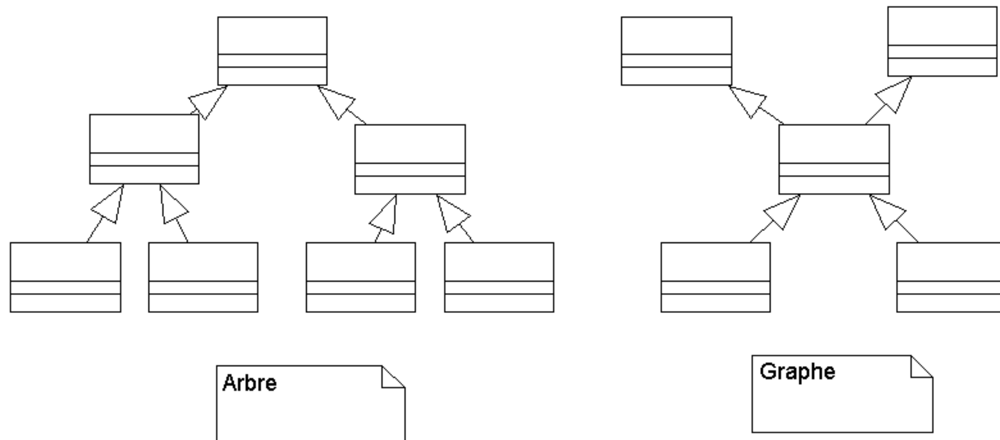


Figure 11-10

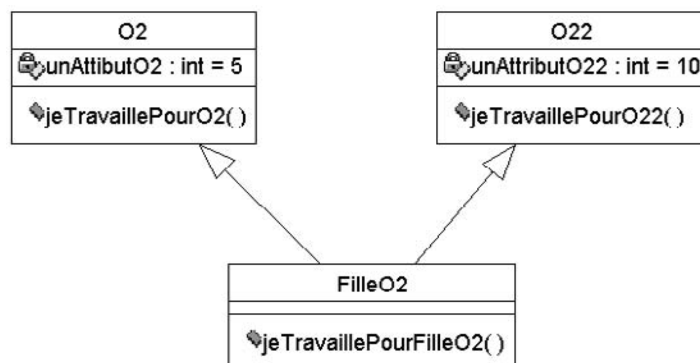
Différence entre un arbre où la ramification ne peut se faire que de manière descendante et un graphe où celle-ci peut se faire dans les deux sens.

Multihéritage en C++ et Python

Nous allons, dans un premier temps, illustrer le multihéritage, en nous limitant au C++ et Python, étant donné son bannissement des autres langages de programmation. Nous poursuivrons uniquement avec ces langages, en découvrant, par une succession de petits exemples, des situations normales et d'autres plus problématiques, ces mêmes situations qui ont incité Java, C# et PHP 5 à préférer s'abstenir. Remarquez, de fait, que, pour réaliser le petit diagramme UML ci-après, nous sommes passés de TogetherJ à Rational Rose, car TogetherJ étant parfaitement synchronisé avec Java, un tel diagramme n'aurait pu être réalisé. Au contraire, la version de Rose utilisée ici est prévue pour s'interfacer avec le C++.

Figure 11-11

Exemple de multihéritage : la classe FilleO2 hérite de deux superclasses.



Le code C++ correspondant est indiqué ci-après.

Code C++ illustrant le multihéritage

```
class O2 {
  private:
    int unAttributO2;
  public:
    O2() {
      unAttributO2 = 5;
    }
    void jeTravaillePourO2() {
      cout << "Je suis un service rendu par la classe O2" << endl;
    }
};
class O22 {

  private:
    int unAttributO22;
  public:
    O22() {
      unAttributO22 = 10;
    }
    void jeTravaillePourO22() {
      cout << "Je suis un service rendu par la classe O22" << endl;
    }
};
class FilleO2 : public O2, public O22 { /* Hérite des deux classes */
  public:
    FilleO2() {}
    void jeTravaillePourLaFilleO2() {
      cout << "Je suis un service rendu par la classe FilleO2" << endl;
    }
};
class O1 {
  private:
    FilleO2* lienFilleO2;
  public:
    O1(FilleO2* lienFilleO2) {
      this->lienFilleO2 = lienFilleO2;
    }
    void jeTravaillePourO1() {
      lienFilleO2->jeTravaillePourO2();  /* message en provenance de la classe O2*/
      lienFilleO2->jeTravaillePourO22(); /* message en provenance de la classe O22*/
      /* notez qu'un tel message aurait été impossible si l'héritage concernant
       ➥la classe O22 avait été déclaré comme protected ou private */
      lienFilleO2->jeTravaillePourLaFilleO2();
    }
};
int main(int argc, char* argv[]) {
  FilleO2* uneFilleO2   = new FilleO2();
  O1* unObjetO1         = new O1(uneFilleO2);
  unObjetO1->jeTravaillePourO1();
  return 0;
}
```

Le résultat attendu est

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O22
Je suis un service rendu par la classe FilleO2
```

Rien de bien compliqué à cela, les caractéristiques de `O2` et celles d'`O22` deviennent ensemble caractéristiques de `FilleO2`. Chaque objet `FilleO2` sera, tout à la fois, un objet `O2` et un objet `O22`. Il en va de même en Python comme le code suivant, équivalent en tout point au précédent, l'illustre :

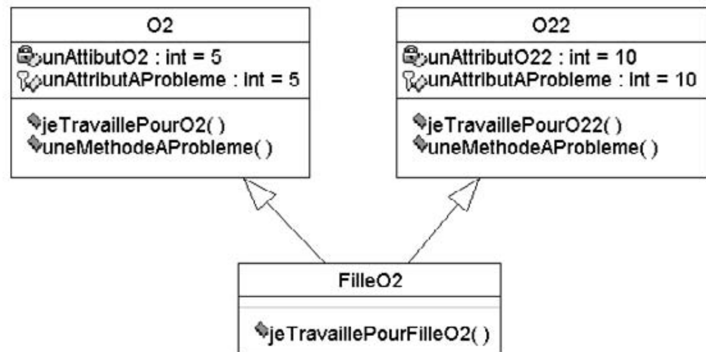
Code Python illustrant le multihéritage

```
class O2:
        __unAttributO2=0
        def __init__(self):
                self.__unAttribut=5
        def jeTravaillePourO2(self):
                print "Je suis un service rendu par la classe O2"

class O22:
        __unAttributO22=0
        def __init__(self):
                self.__unAttributO22=10
        def jeTravaillePourO22(self):
                print "Je suis un service rendu par la classe O22"

class FilleO2(O2,O22): #héritage des deux classes
        def __init__(self):
                pass
        def jeTravaillePourLaFilleO2(self):
                print "Je suis un service rendu par la classe FilleO2"

class O1:
        __lienFilleO2=0
        def __init__(self, lienFilleO2):
                self.__lienFilleO2=lienFilleO2
        def jeTravaillePourO1(self):
                self.__lienFilleO2.jeTravaillePourO2()
                self.__lienFilleO2.jeTravaillePourO22()
                self.__lienFilleO2.jeTravaillePourLaFilleO2()

filleO2=FilleO2()
unObjetO1=O1(filleO2)
unObjetO1.jeTravaillePourO1()
```

Des méthodes et attributs portant un même nom dans des superclasses distinctes

Passons maintenant à une première situation plus délicate, obtenue en rajoutant le même attribut : `unAttribut-AProbleme`, dans les deux superclasses, ainsi que deux méthodes, mais présentant la même signature : `une-MéthodeAProbleme()`. La mise à jour est effectuée dans le diagramme UML et dans le code qui suit. Nous avons délibérément commis un anathème OO, en déclarant l'attribut à problème `protected` dans les deux superclasses, c'est-à-dire directement accessibles dans la sous-classe.

Figure 11-12
Un attribut et une méthode portent le même nom dans deux superclasses distinctes.



Code C++ illustrant un premier problème lié au multihéritage

```
class O2 {
  private:
    int unAttributO2;
  protected:
    int unAttributAProbleme;
  public:
    O2() {
      unAttributO2 = 5;
      unAttributAProbleme = 5;
    }
    void uneMethodeAProbleme() {
      cout << "dans O2 attribut a probleme vaut " <<unAttributAProbleme<<endl;
    }
    void jeTravaillePourO2() {
      cout << "Je suis un service rendu par la classe O2" << endl;
    }
};
class O22 {
  private:
    int unAttributO22;
  protected:
    int unAttributAProbleme;
  public:
    O22() {
      unAttributO22 = 10;
      unAttributAProbleme = 10;

    }
    void uneMethodeAProbleme() {
      cout << "dans O22 attribut a probleme vaut" <<unAttributAProbleme<<endl;
    }
    void jeTravaillePourO22() {
      cout << "Je suis un service rendu par la classe O22" << endl;
    }
};
```

```
class Fille02 : public 02, public 022 {
  public:
    Fille02() {}
    void jeTravaillePourLaFille02() {
      cout << 022 ::unAttributAProbleme << endl ; /* il faut spécifier lequel des deux attributs*/
      022::uneMethodeAProbleme(); /* il faut spécifier laquelle des deux méthodes est utilisée */
      cout << "Je suis un service rendu par la classe Fille02" << endl;
    }
};
class 01 {
  private:
    Fille02* lienFille02;
  public:
    01(Fille02* lienFille02) {
      this->lienFille02 = lienFille02;
    }
    void jeTravaillePour01() {
      lienFille02->jeTravaillePour02();
      lienFille02->jeTravaillePour022();
      lienFille02->jeTravaillePourLaFille02();
      lienFille02->02::uneMethodeAProbleme(); /* il faut, ici aussi, spécifier laquelle des deux
      ➥méthodes est utilisée */
    }
};
int main(int argc, char* argv[]) {
  Fille02* uneFille02= new Fille02();
  01* unObjet01 = new 01(uneFille02);
  unObjet01->jeTravaillePour01();
  return 0;
}
```

Resultat

```
Je suis un service rendu par la classe 02
Je suis un service rendu par la classe 022
dans 022 attribut à probleme vaut 10
Je suis un service rendu par la classe Fille02
dans 02 attribut à probleme vaut 5
```

Un problème survient, car les deux superclasses nomment de la même manière un attribut et une méthode. Lors de l'appel de la méthode et de l'attribut dans la sous-classe, naît une ambiguïté fondamentale, épinglée par le compilateur. De laquelle des deux méthodes et duquel des deux attributs s'agit-il ? Le compilateur ne s'offusquera que si la sous-classe fait un usage explicite de la méthode ou de l'attribut à problème. La seule manière pour éviter que le compilateur ne rechigne est de spécifier, comme vous pouvez le voir dans le code, l'attribut et la méthode en question que l'on souhaite utiliser. Cela se fait très simplement, lors de l'appel, en attachant au nom de l'attribut ou de la méthode celui de la classe, comme vous pourriez le faire avec des fichiers portant un même nom, mais situés dans des répertoires distincts. En effet, il s'agit réellement, à l'échelle des attributs et des méthodes, de préciser le chemin à effectuer pour les retrouver ou bien encore de spécifier leur adresse complète.

En Python

```
class O2:
        __unAttributO2=0
        unAttributAProbleme = 0
        def __init__(self):
                self.__unAttributO2=5
                self.unAttributAProbleme = 5
        def uneMethodeAProbleme(self):
                print "dans O2 attribut à problème vaut %s" %self.unAttributAProbleme
        def jeTravaillePourO2(self):
                print "Je suis un service rendu par la classe O2"

class O22:
        __unAttributO22=0
        unAttributAProbleme = 0
        def __init__(self):
                self.__unAttributO22=10
                self.unAttributAProbleme = 10
        def uneMethodeAProbleme(self):
                print "dans O22 attribut à problème vaut %s" %self.unAttributAProbleme
        def jeTravaillePourO22(self):
                print "Je suis un service rendu par la classe O22"

class FilleO2(O2,O22): #héritage des deux classes
        def __init__(self):
                O2.__init__(self)
                O22.__init__(self)
        def jeTravaillePourLaFilleO2(self):
                O2.uneMethodeAProbleme(self) #manière de choisir la version désirée
                print "Je suis un service rendu par la classe FilleO2"

class O1:
        __lienFilleO2=0
        def __init__(self, lienFilleO2):
                self.__lienFilleO2=lienFilleO2
        def jeTravaillePourO1(self):
                self.__lienFilleO2.jeTravaillePourO2()
                self.__lienFilleO2.jeTravaillePourO22()
                self.__lienFilleO2.jeTravaillePourLaFilleO2()
                self.__lienFilleO2.uneMethodeAProbleme()

filleO2=FilleO2()
unObjetO1=O1(filleO2)
unObjetO1.jeTravaillePourO1()
```

Résultats

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O22
dans O2 attribut à problème vaut 10
Je suis un service rendu par la classe FilleO2
dans O2 attribut à problème vaut 10
```

En Python, il faut aussi rendre moins ambigu l'appel à la méthode, et cela de la manière précisée dans le code. Quant à l'attribut, celui-ci étant d'office un attribut d'instance, Python ne considère qu'une seule et unique occurrence de cet attribut, la dernière, d'où l'apparition des deux « 10 ».

S'agissant des méthodes, le problème n'est pas uniquement qu'elles soient signées de la même façon dans les deux superclasses, mais qu'à signature identique ne corresponde pas un corps d'instructions identiques. Alors que cette signature partagée, en présence d'un corps d'instructions différent, est la base du polymorphisme, lorsque les classes impliquées sont à des niveaux hiérarchiques différents, le problème survient, car les deux classes se trouvent au même niveau. Cette dernière considération mène très logiquement à la réponse trouvée à ce problème par Java, C# et PHP 5.

Ces trois langages ne permettent pas le multihéritage de classe, mais permettent, en revanche, le multihéritage d'interfaces (que nous approfondirons au chapitre 15). En Java, C# et PHP 5, une sous-classe peut hériter d'une classe et d'autant d'interfaces que l'on veut ou juste du nombre d'interfaces souhaité. Rappelez-vous que l'interface se limite à la liste des signatures de méthode et, de fait, en l'absence de corps, ne conduira jamais aux problèmes d'ambiguïté rencontrés en C++ et en Python. Rien n'interdit plusieurs interfaces, héritées par une même sous-classe, de posséder des signatures de méthodes communes, étant donné que les difficultés apparaissent uniquement en présence de corps d'instructions différents. En ce qui concerne les attributs, les interfaces Java autorisent uniquement des attributs « public », « finaux » et « statique » (c'est-à-dire des constantes de classe), mais qu'il vous reste malgré tout à nommer différemment. Les interfaces C# et PHP 5, quant à elles, n'en autorisent simplement pas.

Plusieurs chemins vers une même superclasse

La POO favorisant l'éclatement dans le développement logiciel, la possibilité que deux méthodes, présentes dans des classes différentes, portent le même nom (par exemple, implémentant une fonctionnalité commune) n'est pas nulle, d'où la prudence des autres langages. Le seul recours à cela est d'explicitement différencier leur nom ou de spécifier leur chemin d'accès au moment de l'appel. Un autre problème, encore plus subtil, est posé par le nouveau diagramme UML qui suit.

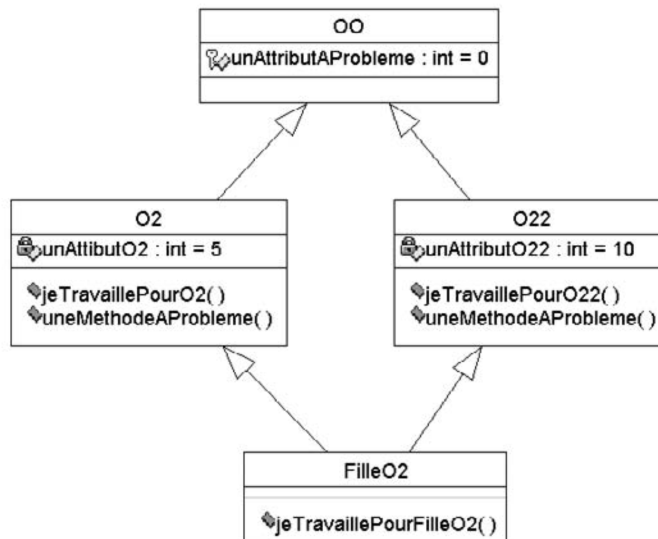


Figure 11-13

Plusieurs chemins d'héritage mènent à une même superclasse.

Code C++ : illustrant un deuxième problème lié au multihéritage

```
class O0 {
  protected:
    int unAttributAProbleme;
  public:
    O0() {
      unAttributAProbleme   = 0;
    }
};
class O2 : virtual public O0 {
  private:
    int unAttributO2;
  public:
    O2() {
      unAttributO2          = 5;
      unAttributAProbleme   = 5;
    }
    void uneMethodeAProbleme() {
      cout << "dans O2 attibut a probleme vaut " <<unAttributAProbleme<<endl;

    }
    void jeTravaillePourO2() {
      cout << "Je suis un service rendu par la classe O2" << endl;
    }
};
class O22 : virtual public O0 {
  private:
    int unAttributO22;
  public:
    O22() {
      unAttributO22         = 10;
      unAttributAProbleme   = 10;
    }
    void uneMethodeAProbleme() {
      cout << "dans O22 attibut a probleme vaut " <<unAttributAProbleme<<endl;
    }
    void jeTravaillePourO22() {
      cout << "Je suis un service rendu par la classe O22" << endl;
    }
};
class FilleO2 : public O2, public O22  /* l'ordre d'héritage va maintenant prendre de l'importance */ {
  public:
    FilleO2() {}
    void jeTravaillePourLaFilleO2() {
      O22::uneMethodeAProbleme();
      cout << "Je suis un service rendu par la classe FilleO2" << endl;
    }
};
```

```
class O1 {
  private:
    Fille02* lienFille02;
  public:
    O1(Fille02* lienFille02) {
      this->lienFille02 = lienFille02;
    }
    void jeTravaillePourO1() {
      lienFille02->jeTravaillePourO2();
      lienFille02->jeTravaillePourO22();
      lienFille02->jeTravaillePourLaFille02();
      lienFille02->O2::uneMethodeAProbleme();
    }
};
int main(int argc, char* argv[]) {
  Fille02* uneFille02   = new Fille02();
  O1* unObjetO1         = new O1(uneFille02);
  unObjetO1->jeTravaillePourO1();
  return 0;
}
```

Le problème qui se pose est le suivant. L'héritage se réalise concrètement par une forme dissimulée de composition, puisque l'objet de la sous-classe possède un objet de la superclasse. Que se passe-t-il quand plusieurs superclasses présentent, elles-mêmes, une superclasse commune, comme dans le cas présent ? Logiquement, tout objet de la classe `Fille02` se composera deux fois d'un objet de la classe OO, une première fois, en provenance de la classe `O2`, une seconde fois de la classe `O22`. Est-ce vraiment nécessaire ? Si on remonte le graphe, des classes plus spécifiques aux classes plus générales, dès qu'une de ces classes est rencontrée en empruntant des chemins différents, le problème se pose. Quand l'héritage n'est pas déclaré « virtuel », la répétition des instances des superclasses dans la sous-classe est la solution par défaut proposée par C++, comme le montre le résultat de l'exécution du code.

Résultat

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O22
dans O22 attibut à problème vaut 10
Je suis un service rendu par la classe Fille02
dans O2 attibut à problème vaut 5
```

L'héritage virtuel

Mais est-ce vraiment un problème ? Il doit y en avoir un, sinon C++ ne vous aurait pas permis de le contourner, en déclarant l'héritage cette fois « virtuel ». Retournons à notre écosystème, les ressources et la faune héritaient toutes deux d'`ObjetJungle`. La Proie n'héritait que de Faune, pourtant toute proie est également une ressource pour le prédateur. Nous pourrions faire hériter la proie et de `Faune` et de `Ressource` comme dans le diagramme ci-après :

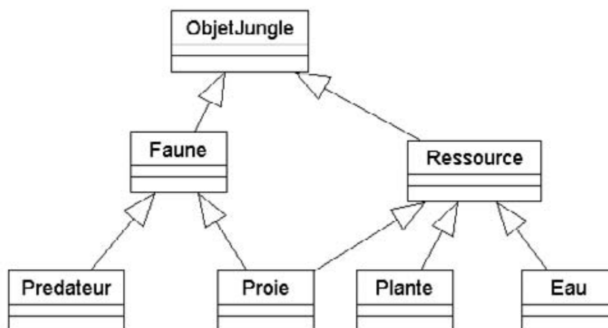


Figure 11-14
Dans ce diagramme de classe illustrant le multihéritage, la proie hérite à la fois de la faune et de ressource.

Est-il nécessaire de dupliquer l'`ObjetJungle`, vu que celui-ci ne contient que des informations sur la position des objets ? Dans ce cas-ci, non bien sûr, car cette information sur la position se doit de rester unique. La solution par défaut du C++ devient inappropriée ici. La seule possibilité consiste à déclarer l'héritage « virtual », avec, pour effet, de rendre toujours unique l'objet de la superclasse partagée par les deux sous-classes. En revanche, un héritage non « virtual », cette fois, resterait approprié dans le cas suivant, illustré également par le petit diagramme qui suit :

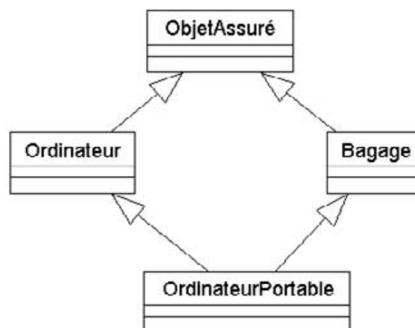


Figure 11-15
Une situation de multihéritage où l'utilisation de l'héritage « virtual » est bénéfique.

Quand on possède une assurance vol pour tous ses ordinateurs et une assurance perte pour tous ses bagages, un ordinateur portable doit pouvoir bénéficier des deux assurances, l'une contre le vol, en tant qu'ordinateur, l'autre contre la perte, en tant que bagage. L'héritage n'est plus virtuel, car il est nécessaire pour information commune aux assurances qu'il soit dupliqué. Quand « virtualiser » l'héritage et quand ne pas le faire ?

Comme nous le voyons, le problème n'est pas simple, et l'accroissement de la difficulté n'a fait que renforcer la conviction de Java, de C# et de PHP 5 d'éviter toutes ces possibles sources d'ambiguïté et de confusion. Comme toujours, C++, quant à lui, nous juge bien plus intelligents que nous ne le sommes en réalité, et nous offre tous les bras de levier et degrés de liberté nécessaires à la bonne décision et à l'optimisation du logiciel résultant. En rendant l'héritage virtuel dans le code précédent, il ne peut plus y avoir qu'une seule instance de l'attribut à problème.

Le résultat sera maintenant le suivant :

Résultat avec héritage virtuel

```
Je suis un service rendu par la classe O2
Je suis un service rendu par la classe O22
dans O22 attribut à problème vaut 10
Je suis un service rendu par la classe FilleO2
dans O2 attibut à problème vaut 10
```

Les « 10 » seraient à remplacer par des « 5 » si on inversait l'ordre de l'héritage.

Le problème se pose également avec Python, dès l'apparition de ce losange dans les relations d'héritage, cependant il ne pose pas pour les attributs mais pour la redéfinition des méthodes, comme nous le verrons dans le chapitre suivant.

Exercices

Exercice 11.1

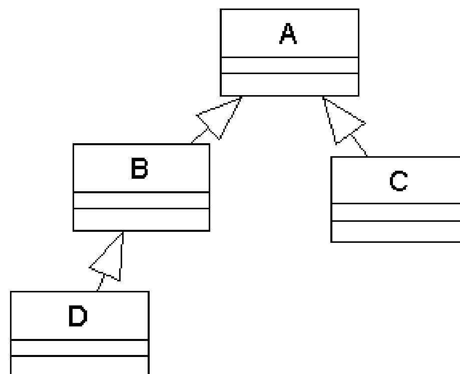
Dessinez un diagramme de classes UML intégrant les classes suivantes : appareil électroménager, appareil à cuisiner, appareil à nettoyer, ramasse-miettes (pas le garbage collector, l'autre), lave-vaisselle, micro-ondes, four.

Exercice 11.2

Dessinez un diagramme de classes UML intégrant les classes suivantes : ordinateur, ordinateur fixe, ordinateur portable, PC, Macintosh, Dell portable, MAC portable Titatium G4. Discutez du possible apport du multihéritage.

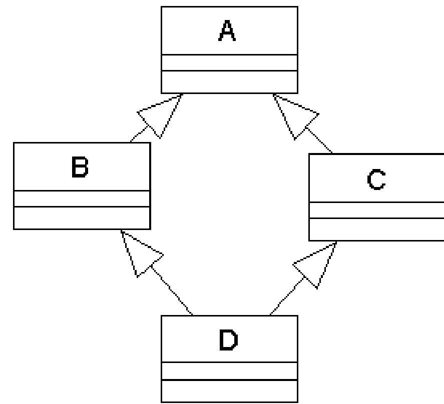
Exercice 11.3

Écrivez le squelette de code dans les trois langages, Java, C# et C++, correspondant au diagramme de classes suivant.



Exercice 11.4

Écrivez le squelette de code dans les trois langages, Java, C# et C++, correspondant au diagramme de classes suivant. Lorsque cela est nécessaire, remplacez les classes par des interfaces.



Exercice 11.5

Aucun des trois petits codes suivant ne trouvera grâce aux yeux des compilateurs. Expliquez pourquoi et corrigez-les en conséquence :

Fichier Exo1.java

```
class O1 {}
class O2 {}
public class Exo1 extends O1,O2 {
  public static void main(String[] args) {
  }
}
```

Fichier Exo2.cs

```
public class O1 {}
public interface O2 {
  int jeTravaillePourInterface();
}
public class Exo2 : O1,O2 {
  public static void Main() {}
}
```

Fichier Exo3.cpp

```
class O1 {
  public:
    void jeTravaillePourLaClasse() {}
};
class O2 {
  public:
    void jeTravaillePourLaClasse() {}
```

```
};
class Fille01 : public 01, public 02 {
  public:
    void jeTravaillePourFille01() {
      jeTravaillePourLaClasse();
    }
};
int main(int argc, char* argv[]) {
  printf("Il y a un probleme\n");
  return 0;
}
```

Exercice 11.6

Aucun des trois petits codes suivants ne trouvera grâce aux yeux des compilateurs. Expliquez pourquoi et corrigez-les en conséquence :

Exo1.java

```
class 01 {}
class 02 extends 01 {}
public class Exo1 {
  public static void main(String[] args) {
    01 un01 = new 01();
    02 un02 = new 02();
    un02    = un01;
  }
}
```

Exo2.cs

```
public class 01 {}
public interface 02 {}
public class 03 : 01, 02 {}
public class Exo2 {
  public static void Main() {
    01 un01 = new 01();
    03 un03 = new 03();
    02 un04 = un03;
    03 un05 = un04;
  }
}
```

Exo3.cpp

```
class 01 {
  public:
    void jeTravaillePourLaClasse() {}
};
class 02 {
};
```

```
class Fille01 : public 01, public 02 {
  public:
    void jeTravaillePourFille01() {
      jeTravaillePourLaClasse();
    }
};
int main(int argc, char* argv[]) {
  02 un02;
  Fille01 unF01;
  unF01= un02;
  printf("Il y a un probleme\n");
  return 0;
}
```

Exercice 11.7

Expliquez pourquoi, malgré l'existence de l'accès `protected` dans les trois langages, la charte du bon programmeur OO vous incite à ne pas l'utiliser.

Exercice 11.8

Quelle différence existe-t-il en C++ entre un héritage `public` et `private`. Pour quelle raison, selon vous, cette subtilité a disparu de Java et C# ?

Exercice 11.9

Quelle version de cette assertion est-elle exacte ?

> « Partout où apparaît un objet d'une superclasse, je peux le remplacer par un objet de sa sous-classe »

ou

> « Partout où apparaît un objet d'une sous-classe, je peux le remplacer par un objet de sa superclasse »

Exercice 11.10

Soit `superA` une classe et `sousA` sa sous-classe, comment le « casting » sera-t-il employé :

```
a = (sousA)b
```

ou

```
a=(superA)b ?
```

Exercice 11.11

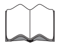
Pourquoi les classes `Stream` et `GUI` en Java se prêtent-elles idéalement à la mise en pratique des mécanismes d'héritage ?

12

Redéfinition des méthodes

Ce chapitre décrit une des possibilités offertes par l'héritage et qui est à la base du polymorphisme : la redéfinition dans les sous-classes de méthodes, d'abord définies dans la superclasse. La mise en œuvre de cette pratique et le résultat surprenant, différent selon les langages, de ces effets, tant pendant la phase de compilation que lors de celle de l'exécution, seront analysés en profondeur.

Sommaire : Redéfinition des méthodes — Polymorphisme — Les mots-clés `super` et `base` et `parent` — Java, Python et PHP 5 : polymorphique par défaut — C++ : non polymorphique par défaut — C# : pas vraiment de défaut — Un mauvais « casting »



Candidus — Peut-on avoir une vision philosophique de l'héritage ? Autrement dit, peut-on en avoir une compréhension telle qu'il soit exploité, naturellement, de façon bien inspirée par le programmeur ?

Doctus — Les langages de programmation montrent des différences qui se situent justement sur ce plan « philosophique ». Je te répondrai donc qu'il est même important de se forger une telle vision de l'héritage, du polymorphisme et de leurs conséquences.

Cand. — Quelles peuvent être les différences d'inspiration de nos cinq langages ?

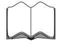
Doc. — Toujours les mêmes soucis de performance et de fiabilité : le compilateur, dans son rôle de juge de la cohérence, doit pouvoir trouver toutes les pièces du puzzle dans le travail du programmeur. Il doit pouvoir constater qu'elles s'assemblent parfaitement. Et l'OO nous permet sans restriction d'utiliser différents objets pour assurer un même rôle. Sa seule exigence, ce sera que l'on puisse s'assurer qu'ils disposent chacun des méthodes correspondantes.

Cand. — Tu penses à une superclasse joker remplacée par une instance effective au moment de l'exécution, n'est-ce pas ?

Doc. — Exactement. Et c'est là que le compilateur, dans les langages qui ont choisi d'y recourir, joue un rôle différent suivant nos langages objet. Ils peuvent donner ou non prépondérance à l'étape de compilation pour guider le déroulement de l'exécution. En d'autres termes, en fonction de l'importance qu'ils donnent au type déclaré des objets par rapport au type qu'ils endossent au moment de l'exécution.

Cand. — Je conçois effectivement que si un programme s'amuse à construire le puzzle lors de l'exécution, sa performance en prend un coup ! Mais une chose m'intrigue dans le remplacement dynamique des jokers : les informations exigées par le compilateur C++ consistent à demander au programmeur de lever de possibles ambiguïtés. Qu'en est-il pour Java qui, à l'exécution, va faire le même travail sans l'assistance du programmeur ?

Doc. — Pas de magie, encore une fois. Un mécanisme d'exceptions sera mis en jeu. Le programmeur devra donc prévoir du code pour traiter ces situations, sinon le programme s'interrompra. Le gain n'est effectif que si ces accidents sont rares...



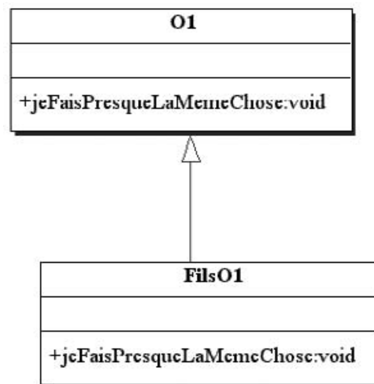
La redéfinition des méthodes

Nous avons vu que l'héritage permet à des classes d'être à la fois elles-mêmes, et en même temps un ensemble successif de superclasses. Elles sont elles-mêmes car, en plus des caractéristiques qu'elles héritent de leur parent (avec « s » ou sans « s »), grand-parent et arrière-grand-parent, elles peuvent rajouter des attributs et des méthodes qui leur sont propres. L'héritage permet également un mécanisme supplémentaire, un tant soit peu plus subtil, mais extrêmement puissant : la redéfinition de méthodes déjà existantes chez le ou les parents.

Comme indiqué dans le petit diagramme UML ci-après, il s'agit de récupérer la même signature de méthode que celle déclarée chez le père, mais d'en modifier le corps d'instruction. En substance, la classe père et la classe fils partagent une activité, bien qu'elles l'exécutent différemment. Ce qu'elles partagent en réalité, c'est le nom de l'activité, mais pas la pratique à proprement parler. Dans le code Java correspondant à ce diagramme, on constate que le corps d'instructions de la version du fils de cette méthode partagée avec le père fait d'abord appel à la version du père, avant d'y mettre son grain de sel. Le mot-clé `super` sert simplement de référent vers la superclasse. En son absence, on se serait retrouver en présence d'une dangereuse boucle récursive infinie. Il est, de ce fait, indispensable, afin de préciser la version de la méthode dont il s'agit : celle du fils ou celle du père. C'est un type d'écriture très souvent rencontré, pour des raisons que nous expliquerons plus avant.

Figure 12-1

Redéfinition dans la sous-classe « FilsO1 » de la méthode jeFaisPresqueLaMemeChose() définie originellement dans la superclasse « O1 ».




```
public class O1 {
  public void jeFaisPresqueLaMemeChose() {
  }
}
public class FilsO1 extends O1 {
  public void jeFaisPresqueLaMemeChose() {
    super.jeFaisPresqueLaMemeChose();
    ..................................
  }
}
```

Tel père tel fils. C'est une réalité que celle des fils cherchant à imiter leur père. Chanteur, acteur, écrivain, sportif, homme d'affaires ou politique, comme papa, il devra s'éloigner de celui-ci pour enfin devenir un artiste hors père. Pourquoi l'application de ce principe de redéfinition des méthodes, participant de l'héritage, est-elle très courante en OO ?

Beaucoup de verbiage mais peu d'actes véritables

L'héritage s'est inspiré de nos mécanismes d'organisation cognitifs, pour transposer les avantages qu'ils permettent : simplicité, économie, adaptabilité, flexibilité, réemploi, au développement logiciel. Une autre caractéristique de notre conceptualisation du monde est que nous consacrons moins de concepts à en décrire les propriétés fonctionnelles et actives que les simples propriétés structurelles, comme si l'image que nous nous faisions de la nature était moins riche en fonctionnalité qu'en structure.

Le vocabulaire que nous dédions aux modalités actives est moins riche que celui dédié à la perception statique des choses. C'est d'ailleurs une des raisons fondamentales qui expliquent que nous organisions notre conceptualisation de manière taxonomique : nous regroupons toutes les classes qui partagent les mêmes modalités actives. Tous les animaux, les millions d'espèces existantes, vivent, mangent, dorment et meurent. Ils le font sans doute d'une manière qui leur est propre, mais ils le font tous. Les chanteurs d'opéra, de rock, de folk, de jazz, de gospel, ceux à la croix de bois…, chantent tous, font tous des disques, passent à la télé mais, heureusement pour nous, de façon différente et pas en même temps.

Il n'est dès lors pas surprenant de retrouver des mêmes noms d'activité, ici de méthodes, pour les classes et leurs sous-classes. Notez que cette mise en commun des noms d'activités à différents niveaux hiérarchiques prend toute sa raison d'être, tant dans la pratique cognitive qu'en programmation, dans des situations où ces activités sont mises en pratiques par une tierce « entité ». Ainsi, dans l'exemple que nous avons vu au premier chapitre, le feu rouge envoie un message unique, « démarre », à tous les véhicules lui faisant face, sans se préoccuper outre mesure de la manière ultime dont ce message sera exécuté par les différents types de véhicule. Que lui importe, en effet, au feu rouge, de savoir comment son message sera perçu par les voitures. Toutes les voitures démarrent, c'est la seule chose qui compte vraiment ! Cette possibilité offerte à une classe d'interagir avec un ensemble d'autres classes, en leur envoyant un même message, compris par toutes mais exécuté de manière différente, explique pour une grande part que l'on retrouve ce message à plusieurs niveaux. Elle est illustrée par le petit diagramme UML qui suit.

Dans le diagramme de classe qui suit, un objet de la classe `O2` déclenche le même message sur tous les objets issus de la superclasse `O1`, mais ce même message sera exécuté différemment en fonction de la sous-classe finale dont est issu l'objet recevant ce message. Une classe peut donc interagir avec un ensemble d'autres comme s'il s'agissait d'une seule et même classe. Elle n'a pas nécessairement besoin d'en connaître la nature ultime pour en disposer. En fait tout un large pan du programme lui devient invisible. C'est une nouvelle forme d'encapsulation si chère à l'OO. La « tierce classe » devient complètement aveugle aux spécifications des différentes sous-classes avec lesquelles elle interagira en dernier ressort. Ces spécifications constitueront ainsi pour le programmeur un large espace de liberté et de variabilité.

Ce faisant, nous entrons en plein dans la pratique du « polymorphisme », que nous retrouverons encore dans les chapitres qui suivent. Ce message, au niveau de la superclasse, possédera, oui ou non, un corps d'instructions par défaut. Nous verrons que, dans un type particulier de classe (appelée classe « abstraite »), le message pourra se borner à n'exister qu'en tant que seule signature. Une sous-classe, au moins, deviendra indispensable afin d'en permettre une première réalisation.

La base du polymorphisme

L'héritage offre la possibilité pour une classe de s'adresser à une autre, en sollicitant de sa part un service qu'elle est capable d'exécuter de mille manières différentes, selon que ce service, nommé toujours d'une seule et même façon, se trouve redéfini dans autant de sous-classes, toutes héritant du destinataire du message. C'est la base du polymorphisme. Cela permet à notre première classe de traiter toutes ses classes interlocutrices comme une seule, et de ne modifier en rien son comportement si on ajoute une de celles-ci ou si une de celles-ci modifie sa manière de répondre aux messages de la première : simplicité de conception, économie d'écriture et évolution sans heurt : tout l'OO est là, dans le polymorphisme.

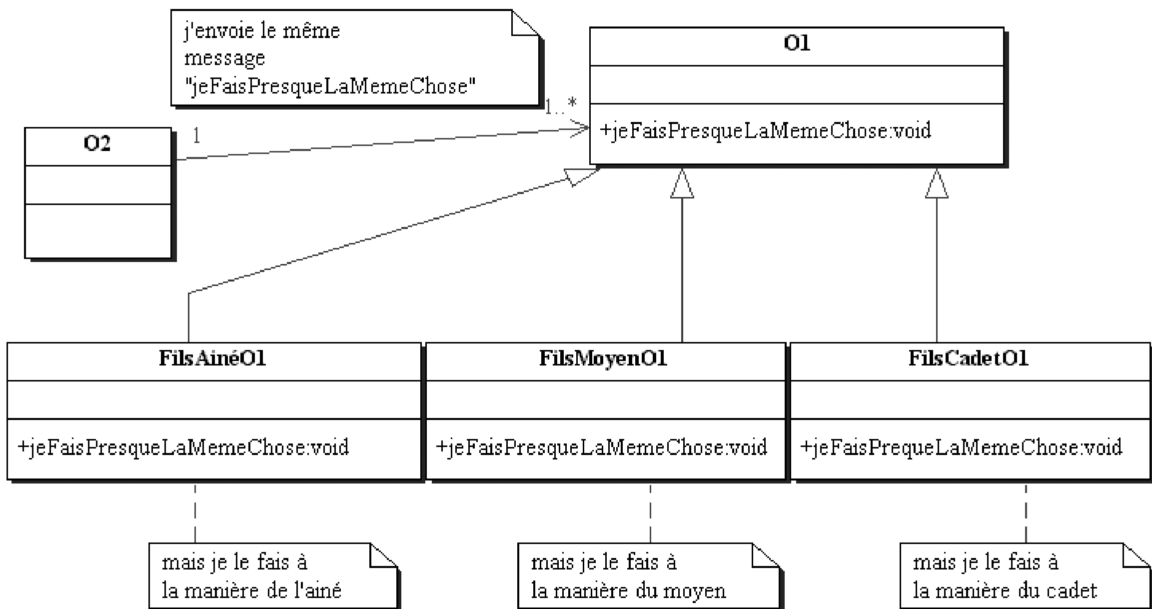


Figure 12-2

Diagramme de classe représentant le polymorphisme. La classe O2 déclenche sur la classe O1 le message jeFaisPresqueLaMemeChose qui se trouve redéfini dans trois sous-classes.

Un match de football polymorphique

Nous allons illustrer ce mécanisme en nous repenchant sur notre simulation du match de football que nous avions juste esquissée dans le chapitre 10, consacré à l'UML. Dans un premier temps, nous avions délibérément évité toute mise en pratique de l'héritage. Or, si une classe se prête assez naturellement à cette mise en pratique, c'est bien la classe `Joueur`, comme montré dans le diagramme qui suit. Nous spécialisons la classe `Joueur` en trois sous-classes : `Attaquant`, `Defenseur` et `Gardien`. Rien, du côté des attributs, ne particularise vraiment les différentes sous-classes de joueur, sans doute une tenue un peu différente pour le gardien de but.

Y a-t-il lieu de rajouter de nouvelles méthodes dans les sous-classes ? Là encore, le gardien de but peut, sans essuyer de punitions de la part de l'arbitre, attraper la balle avec les mains. Le mode d'interaction entre les

joueurs et la balle sera donc quelque peu différent dans le cas du gardien qui peut, dans un premier temps, faire comme tous les joueurs, c'est-à-dire lui donner de violents coups de pied, mais, de surcroît, la caresser de ses douces mains.

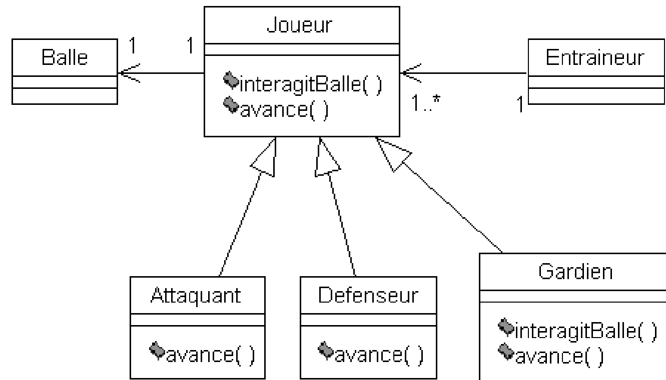


Figure 12-3
Un petit diagramme de classe centré sur les joueurs et leur relation à l'entraîneur.

Dans ce diagramme UML, très simplifié, aucun nouvel attribut ni méthode ne vient se rajouter dans les sous-classes de joueur. Ce qu'octroie l'héritage ici est la redéfinition des méthodes : `interagitBalle()` pour le gardien de but, et `avance()` pour tous les joueurs. Pour illustrer le polymorphisme de la méthode `avance()`, le scénario imaginé, est un entraîneur excité et paniqué en fin de partie, qui hurle d'avancer à tous ses joueurs. C'est son envoi de message à lui. Sans doute le dernier avant longtemps. Chaque joueur va donc avancer, mais tous le feront à leur manière. Surtout, dans ce hurlement de la dernière chance, il n'est plus question pour l'entraîneur d'en particulariser le contenu en fonction des joueurs auxquels il est adressé. Que ceux-ci exécutent à leur manière les ordres, ainsi qu'ils ont appris à le faire pendant les entraînements.

On a rarement vu, sauf dans des cas vraiment désespérés, le gardien de but se retrouver à flirter avec son *alter ego* de l'équipe adverse. En fait, tous les joueurs se déplaceront, mais en respectant une zone de déplacement sur le terrain, liée à la place qu'ils occupent ainsi qu'au placement des joueurs adverses. Bel exemple de polymorphisme. Nous allons d'ailleurs illustrer cette première mise en musique du polymorphisme dans les cinq langages de programmation, et de manière très graduelle, afin d'en expliquer les avantages, les subtilités syntaxiques, et, là encore, les écarts commis par le C++. De nouveau, ce dernier a pris le pli de faire les choses de manière plus compliquée que les autres.

La classe Balle

Commençons d'abord par la classe la moins problématique :

En Java

```
class Balle {
  public Balle() {}
  public void bouge(){
   System.out.println("la balle bouge");
  }
}
```

En C++

```
class Balle {
public:
 Balle() {}
  void bouge(){
   cout <<"la balle bouge"<<endl;
  }
}
```

En C#

```
class Balle{
  public Balle() {}
  public void bouge(){
   Console.WriteLine("la balle bouge");
  }
}
```

En Python

```
class Balle:
     def __init__(self):
                pass
     def bouge(self):
                print "la balle bouge"
```

En PHP 5

```
class Balle {
     public function __construct() {}
     public function bouge() {
                print ("la balle bouge <br> \n");
     }
}
```

Passons maintenant à une classe plus sensible, la classe `Joueur` :

En Java

```
class Joueur{
 private int posSurLeTerrain;
 private Balle laBalle;
 public Joueur(Balle laBalle) {
  this.laBalle  = laBalle;
 }
 public int getPosition() {
  return posSurLeTerrain;
 }
 public void setPosition(int position) {
  posSurLeTerrain = position;
 }
```

```
  public void interagitBalle() {
   System.out.println("Je tape la balle avec le pied");
   laBalle.bouge();
  }
  public String toString() // redéfinition de la méthode toString() définie dans la classe Object
  {
    return getClass().getName() ;
  }
  public void avance() {
   System.out.println("la position actuelle du " + this + " est " + posSurLeTerrain);
   /* this déclenche automatiquement l'appel de toString() pour obtenir la classe ultime
      de l'objet */
   posSurLeTerrain += 20;
  }
 }
```

`posSurLeTerrain` est un attribut clé qui indiquera la position du joueur à un instant donné sur le terrain. Dans un souci de simplicité, on ne spécifie qu'une valeur, qui pourrait être la distance par rapport au but. C'est la valeur maximale de cette distance, selon la nature des joueurs, qui imposera de redéfinir leur déplacement. Comme nous utiliserons quelques fois cet attribut dans les sous-classes à venir, des méthodes d'accès, `get()` et `set()`, deviennent nécessaires.

Une autre addition, qui illustre parfaitement le thème principal de ce chapitre, est la redéfinition de la méthode `toString()`. Cette méthode existe par défaut dans la classe la plus haute de la hiérarchie Java, la classe `Object`, que nous retrouverons dans le chapitre 14 et dont par défaut, sans que l'on doive l'expliciter, hérite toute classe Java. Elle est appelée implicitement à chaque fois que l'on demande d'afficher le référent d'un objet objet (dans le code par la présence du `this`). Comme il nous importe d'afficher la classe dynamique de l'objet dans le corps de la méthode `avance()`, nous redéfinissons la méthode `toString()` afin qu'elle fasse précisément ceci : renseigner la classe dynamique de l'objet par le concours des deux méthodes introspectives `getClass().getName()`. Enfin, en plus de récupérer et d'afficher la classe de l'objet en question, à chaque exécution de la méthode `avance()`, le joueur incrémente sa position de 20.

En C++

```
class Joueur {
 private:
  int posSurLeTerrain;
  Balle* laBalle;
 public:
  Joueur(Balle* laBalle) {
   this->laBalle = laBalle;
  }
  /*virtual*/ void interagitBalle() { /* présence de "virtual" */
   cout<<"Je tape la balle avec le pied" << endl;
   laBalle->bouge();
  }
  /*virtual*/ void avance() { /* présence de "virtual" */
   const type_info& t = typeid(*this);  /* pour faire apparaître la classe de l'objet */
   cout<<"la position actuelle du " << t.name() << " joueur est " << posSurLeTerrain << endl;
   posSurLeTerrain += 20;
  }
```

```
  int getPosition() {
   return posSurLeTerrain;
  }
  void setPosition(int position) {
   posSurLeTerrain = position;
  }
};
```

Plusieurs points doivent être détaillés dans la version C++. Tout d'abord, vous voyez apparaître un étrange mot-clé `virtual`, au début de la signature des méthodes qui vont se prêter à une redéfinition dans les sous-classes. Ce mot-clé marque une différence essentielle dans la manière dont les langages qui nous occupent abordent le polymorphisme, et que nous commenterons plus longuement par la suite. Dans un premier temps, nous laisserons ce mot-clé inactif (entre commentaires). La méthode `avance()` est un peu plus compliquée, car nous voulons, à l'instar du `toString()` en Java, récupérer, pendant l'exécution, la classe de l'objet sur lequel la méthode `avance()` s'exécute. C'est un type d'information, caractéristique de RTTI (Run-Time Type Information), assez récemment ajouté dans le fonctionnement du C++ (celui-ci n'était pas conçu pour fonctionner en mode polymorphique). Son utilisation est un peu délicate. Il vous faut en comprendre l'utilité sans nécessairement maîtriser sa syntaxe assez sibylline.

En C#

```
class Joueur {
 private int posSurLeTerrain;
 private Balle laBalle;

 public Joueur(Balle laBalle) {
  this.laBalle = laBalle;
 }

 public /*virtual*/ void interagitBalle() { /* le même " virtual " qu'en C++ */
  Console.WriteLine("Je tape la balle avec le pied");
  laBalle.bouge();
 }

 public /*virtual*/ void avance() { /* toujours virtual */
  Console.WriteLine("la position actuelle du " + this + " est " + posSurLeTerrain);
  posSurLeTerrain += 20;
 }

 public int positionGet { /* remarquez encore la nature singulière des méthodes d'accès */
  get {
   return posSurLeTerrain;
  }
  set {
   posSurLeTerrain = value ; /* « value » sera remplacé par n'importe quelle valeur que l'on passe à
                               l'attribut au moment de l'appel de cette méthode */
  }
 }
}
```

À nouveau, en C#, on note l'apparition de cet étrange `virtual` (pour les mêmes raisons qu'en C++) au début de la déclaration des méthodes, qui sera commenté par la suite. On relève à part cela quelques petites différences syntaxiques. Ainsi aurez-vous pu noter par vous-même l'utilisation des majuscules plutôt que des minuscules pour les méthodes (`Main()` et `non main()`, `ToString()` et `non toString()`). C'est toujours le cas et cela mériterait sans doute un procès ! La méthode `ToString()`, qui provient ici aussi de la superclasse Object dont héritent par défaut toutes les classes, n'a nul besoin d'une redéfinition ici car, dans sa version d'origine (celle dans la classe `Object`), elle fait ce qu'on souhaite qu'elle fasse, c'est-à-dire juste renvoyer la classe dynamique de l'objet en question. Toutefois, c'est également une méthode qui prête souvent à redéfinition de manière en tout point semblable à celle du code Java.

Plus original, comme nous l'avons déjà vu, est la manière dont C # réalise les méthodes d'accès `get` et `set`. Il les réunit dans une même méthode, avec la syntaxe quelque peu singulière indiquée dans le code. Cette subtilité d'écriture permet d'appeler les méthodes d'accès comme s'il s'agissait directement des simples attributs. Par la présence de `virtual`, mis comme commentaire pour l'instant, vous aurez pu constater que C# n'est pas exactement à Java ce que Canada Dry est à l'alcool puisque, de temps en temps, comme ici, il lui fausse compagnie pour se rapprocher du C++.

En Python

```
class Joueur(object): # ici il faut explicitement hériter de la supersuperclasse object
   __posSurLeTerrain=0
   __laBalle=None
   def __init__(self,laBalle):
            self.__laBalle=laBalle
   def getPosition(self):
            return self.__posSurLeTerrain
   def setPosition(self,position):
            self.__posSurLeTerrain=position
   def interagitBalle(self):
            print "Je tape la balle avec le pied"
            self.__laBalle.bouge()
   def __str__(self): # redéfinition de cette méthode
            return self.__class__.__name__
   def avance(self):
            print "la position actuelle du " + self.__str__()+" est %s" %(self.__posSurLeTerrain)
            self.__posSurLeTerrain+=20
```

En Python, nous redéfinissons, comme en Java et en C#, la méthode de description des référents, ici la méthode `__str__()`. Cependant, une différence importante avec les deux langages précédents est l'obligation d'expliciter l'héritage de la superclasse `object`. De manière générale, Python ne fait rien de gratuit et vous oblige à déclarer toutes les initiatives que vous prenez, y compris celles qui pourraient apparaître automatiques à la plupart des programmeurs.

En PHP 5

```
class Joueur {
      private $posSurLeTerrain;
      private $laBalle;
```

```
    public function __construct($laBalle) {
        $this->posSurLeTerrain = 0;
        $this->laBalle = $laBalle;
    }

    public function getPosition() {
        return $this->posSurLeTerrain;
    }

    public function setPosition ($position) {
        $this->posSurLeTerrain = $position;
    }

    public function interagitBalle() {
        print ("Je tape la balle avec le pied <br> \n");
        $this->laBalle->bouge();
    }

    public function __toString () {
        return get_class($this);
    }

    public function avance() {
        print ("la position actuelle du ");
        print ($this);
        print (" est $this->posSurLeTerrain <br> \n");
        $this->posSurLeTerrain += 20;
    }
}
```

On retrouve un code très proche de Java et de C#.

Précisons la nature des joueurs

Attaquons maintenant le principal sujet de ce chapitre, à savoir l'héritage et, surtout, la redéfinition des méthodes dans les sous-classes. Trois sous-classes : `Gardien`, `Defenseur` et `Attaquant` vont hériter de la classe `Joueur`. Dans la classe `Gardien`, les deux méthodes `interagitBalle()` et `avance()` seront redéfinies alors que, dans les deux autres sous-classes, seule la méthode `avance()` le sera.

Redéfinir une méthode dans une sous-classe (ce qu'en anglais on désigne comme la pratique `override`) consiste à reprendre exactement sa signature, à ceci près que l'on rend l'accès à la méthode dans la sous-classe moins sévère qu'il ne l'est dans la superclasse. Un `public` ou `protected` peut remplacer un `private`, mais non l'inverse, par simple respect du principe de substitution. Une superclasse ne peut en faire plus qu'une sous-classe, pas plus qu'une méthode dans une sous-classe ne peut se rendre moins accessible que celle qu'elle redéfinit dans la superclasse. Si je peux envoyer un message à tous les objets issus d'une superclasse, je dois pouvoir envoyer ce même message à tous les objets issus de la sous-classe de celle-ci. *A priori*, une méthode définie comme `private` dans la superclasse, étant inaccessible et de l'extérieur et par ses enfants, ne devrait jamais pouvoir se prêter à une redéfinition. Elle ne le sera pas en effet et nous reviendrons sur ce point précis en fin de chapitre.

La redéfinition des méthodes

Une méthode redéfinie dans une sous-classe possédera la même signature que celle définie dans la superclasse avec, comme unique différence possible, un mode d'accès moins restrictif. En général, une méthode définie `protected` ou `public` dans la superclasse sera redéfinie comme `protected` ou `public` dans la sous-classe.

En Java

```
class Gardien extends Joueur { /* héritage */
 public Gardien(Balle laBalle) {
  super(laBalle); /* appel du constructeur de la superclasse */
  setPosition(0);
 }
 public void interagitBalle() { /* redéfinition */
  super.interagitBalle(); /* appel de la méthode originelle */
  System.out.println("Je prends la Balle avec les mains");
 }
 public void avance() { /* redéfinition */
  if (getPosition() < 10)
   System.out.println("Moi gardien, je peux encore prendre la balle avec les mains");
  if (getPosition() < 20)
   super.avance(); /* appel de la méthode originelle sous condition */
 }
}

class Defenseur extends Joueur {
 public Defenseur(Balle laBalle) {
  super(laBalle);
  setPosition(20);
 }
 public void avance() {
  if (getPosition() < 100)
   super.avance();
 }
}

class Attaquant extends Joueur {
 public Attaquant (Balle laBalle) {
  super(laBalle);
  setPosition(100);
 }
 public void avance() {
  if (getPosition() < 200) {
   super.avance();
   if (getPosition() > 150)
    System.out.println("moi attaquant je fais attention au hors-jeu") ;
  }
 }
}
```

Tout d'abord, penchons-nous sur le constructeur de `Gardien`. Celui-ci fait appel, par l'entremise de `super()`, au constructeur de la superclasse. Rappelez-vous que `super` pointe vers la superclasse. Nous avons vu ce principe dans le chapitre précédent, chaque classe s'occupe de l'initialisation de ses propres attributs. Comme l'attribut `laBalle` trouve son origine dans la superclasse `Joueur`, c'est automatiquement le constructeur de celle-ci (pour autant qu'il s'en occupait déjà dans la superclasse) qui devra à nouveau prendre en charge son initialisation dans la sous-classe

Passons maintenant à la redéfinition des deux méthodes. La méthode `interagitBalle()`, redéfinie seulement chez le gardien, se comporte, dans un premier temps, comme la méthode déclarée initialement chez le `Joueur` (et c'est de nouveau la raison de l'utilisation du mot-clé `super`, indispensable afin d'éviter une récursion infinie), mais ensuite rajoute une fonctionnalité qui lui est propre : « prendre la balle avec les mains ». La méthode `avance ()`, quant à elle, ne se produira que dans des limites permises par la fonction et le placement de chacun des joueurs.

Ici, l'idée est plutôt de renforcer l'intégrité des sous-classes par rapport à la superclasse, en interdisant à l'attribut `posSurLeTerrain` de prendre toutes les valeurs possibles. Conditionner dans la rédéfinition de la méthode l'appel à la méthode originale est également une pratique assez courante. Une sous-classe a quelquefois ceci de plus spécifique que ses attributs, au contraire de ce qui se passe pour la superclasse, ne peuvent prendre toutes les valeurs. Cela colle parfaitement à la vision « ensembliste » de l'héritage, puisque seul un sous-ensemble de toutes les valeurs d'attributs possibles sera admis pour les sous-classes.

En C++

```
class Gardien : public Joueur {
 public:
  Gardien(Balle* laBalle):Joueur(laBalle) {  /* appel du constructeur de la superclasse*/
   setPosition(0);
  }
  void interagitBalle() {           /* redéfinition */
   Gardien::interagitBalle();        /* appel de la méthode originelle */
   cout<<"Je prends la Balle avec les mains"<<endl;
  }
  void avance() {               /* redéfinition */
   if (getPosition() < 10)
    cout<<"Moi gardien, je peux encore prendre la balle avec les mains" << endl;
   if (getPosition() < 20)
    Joueur::avance();
  }
};
class Defenseur : public Joueur {
 public:
  Defenseur(Balle* laBalle):Joueur(laBalle) {
   setPosition(20);
  }
  void avance() {
   if (getPosition() < 100)
    Joueur::avance();
  }
};
```

```
class Attaquant : public Joueur {
 public:
  Attaquant (Balle* laBalle) : Joueur(laBalle) {
   setPosition(100);
  }
  void avance() {
   if (getPosition() < 200) {
    Joueur::avance();
    if (getPosition() > 150) {
     cout<<"moi attaquant je fais attention au hors-jeu" << endl;
    }
   }
  }
};
```

L'écriture du constructeur contraste assez largement avec la version Java. Le rappel du constructeur de la superclasse, pour les mêmes raisons que celles évoquées pour le code Java, se fait, non plus dans le bloc d'instructions du constructeur, mais, plus directement, à même la déclaration de la signature. Par ailleurs, en C++, `super` n'existe pas, et pour cause, le multihéritage l'interdit. Qui serait le super parmi tous les candidats possibles ? Comme pour la désambiguïsation parfois nécessaire, suite à un multihéritage malheureux, l'appel aux méthodes des superclasses se fait donc par une évocation explicite des classes dont elles proviennent.

En C#

```
class Gardien : Joueur {
 public Gardien(Balle laBalle):base(laBalle) { /* appel du constructeur de la superclasse */
  positionGet = 0;
 }
 public /*override*/ new void interagitBalle() { /* override ou new */
  base.interagitBalle(); /* appel de la méthode originelle */
  Console.WriteLine("Je prends la balle avec les mains");
 }
 public /*override*/ new void avance() { /* override ou new */
  if (positionGet < 10)
   Console.WriteLine("Moi gardien, je peux encore prendre la balle avec les mains");
  if (positionGet < 20)
   base.avance(); /* appel de la méthode originelle */
 }
}
class Defenseur : Joueur {
 public Defenseur(Balle laBalle):base(laBalle) {
  positionGet = 20;
 }
 public /*override*/ new void avance() {
  if (positionGet < 100)
   base.avance();
 }
}
class Attaquant : Joueur {
 public Attaquant (Balle laBalle):base(laBalle) {
  positionGet = 100;
 }
```

```
public /*override*/ new void avance() {
 if (positionGet < 200) {
  base.avance();
  if (positionGet > 150)
   Console.WriteLine("moi attaquant je fais attention au hors-jeu");
 }
}
```

En C#, et en ce qui concerne le constructeur, on trouve une pratique hybride de la version Java (avec l'utilisation du mot-clé `base` en lieu et place de `super`), et de C++ (avec l'appel du constructeur de la superclasse lors de déclaration de la signature, plutôt que dans le corps de la méthode). On retrouve d'ailleurs ce même mot-clé `base` dans le corps des méthodes redéfinies.

Vous constaterez également que la signature des méthodes redéfinies inclut le mot-clé `new`. Nous n'avons pas le choix, là encore, sous le regard coercitif du compilateur. Ne rien mettre, comme en Java, provoquerait cette fois un avertissement de la part du compilateur. Il s'agit donc, ou de déclarer les méthodes comme `new` (c'est en effet une nouvelle version de la « même » méthode), ou, alternativement, d'opter pour un couplage du mot-clé `virtual`, lors de la déclaration de la version première de la méthode, avec le mot-clé `override`, lors de la redéfinition de la méthode. Bien évidemment, l'effet n'est pas le même, comme nous allons le constater très bientôt.

En Python

```
class Gardien(Joueur):
       def __init__(self,laBalle):
               Joueur.__init__(self,laBalle)
               self.setPosition(0)
       def interagitBalle(self):
               Joueur.interagitBalle(self)
               print "Je prends la Balle avec les mains"
       def avance(self):
               if self.getPosition()<10:
                      print "Moi gardien, je peux encore prendre la balle avec les mains"
               if self.getPosition()<20:
                      Joueur.avance(self)

class Defenseur(Joueur):
       def __init__(self,laBalle):
               Joueur.__init__(self,laBalle)
               self.setPosition(20)
       def avance(self):
               if self.getPosition()<100:
                      Joueur.avance(self)

class Attaquant(Joueur):
       def __init__(self,laBalle):
               Joueur.__init__(self,laBalle)
               self.setPosition(100)
       def avance(self):
               if self.getPosition()<200:
                      Joueur.avance(self)
               if self.getPosition()>150:
                      print "moi attaquant je fais attention au hors-jeu"
```

En Python, point de `base` et de `super`, mais comme en C++, il est obligatoire de préciser de quelle superclasse provient la méthode que nous exploitons à la redéfinition de la méthode de la sous-classe.

En PHP 5

```
class Gardien extends Joueur {
	public function __construct($laBalle) {
		parent::__construct($laBalle); // appel du constructeur de la superclasse
		$this->setPosition(0);
	}

	public function interagitBalle() {
		parent::interagitBalle();
		print ("Je prends la Balle avec les mains <br> \n");
	}

	public function avance() {
		if ($this->getPosition() < 10) {
			print ("Moi gardien, je peux encore prendre la balle avec les mains <br> \n");
		}
		if ($this->getPosition() < 20) {
			parent::avance();
		}
	}
}

class Defenseur extends Joueur {
	public function __construct($laBalle) {
		parent::__construct($laBalle);
		$this->setPosition(20);
	}

	public function avance() {
		if ($this->getPosition() < 100) {
			parent::avance();
		}
	}
}

class Attaquant extends Joueur {
	public function __construct($laBalle) {
		parent::__construct($laBalle);
		$this->setPosition(100);
	}

	public function avance() {
		if ($this->getPosition() < 200) {
			parent::avance();
	}
	if ($this->getPosition() > 150) {
		print ("Moi attaquant je fais attention au hors-jeu <br> \n");
		}
	}
}
```

Le code PHP 5 est très proche du Java et C# avec présence du mot-clé `parent` jouant un rôle équivalent au `super` de Java et `base` de C#. Cela a pratiquement épuisé toutes les possibilités sémantiques pour ce mot-clé. Que reste-t-il pour les langages à venir : « tonton » ou « maître » ?

Passons à l’entraîneur

… avant qu’une crise cardiaque ne l’éloigne à jamais des terrains de football. Ils n’ont vraiment aucune classe, ces entraîneurs.

En Java

```
class Entraineur{
  private Joueur[] lesJoueurs;
  public Entraineur(Joueur[] lesJoueurs){
   this.lesJoueurs = lesJoueurs;
  }
  public void panique(){
   System.out.println("C'est la panique");
   for (int i=0; i<lesJoueurs.length; i++){
     lesJoueurs[i].avance() ; // le même message à tous les joueurs – attention !!! polymorphisme
   }
  }
}
```

Rien de bien spécial, si ce n’est, aspect capital et clé du polymorphisme, que l’entraîneur est associé à un tableau d’objets typé `Joueur`, ce qui se traduit par une relation 1 -> 1..n , apparaissant entre la classe `Entraineur` et la classe `Joueur` dans le diagramme de classe UML de la figure 12-3.. De son seul point de vue, tous les objets avec lesquels l’entraîneur se doit d’interagir sont issus de la classe `Joueur`. Il ne voit aucun gardien, attaquant ou défenseur parmi eux. On pourrait rajouter les sous-classes `Avant-Centre` ou `Ailier-Droit` qu’il n’en ferait aucun cas. C’est dans sa méthode `panique()` que l’entraîneur envoie le message désespéré d’avancer à tous les joueurs. L’entraîneur envoie ce message à tous les joueurs de son tableau, sans se préoccuper d’aucune sorte de la nature du joueur qui le recevra. Sa seule certitude (le compilateur le lui a assuré) c’est que tous ses joueurs seront en mesure de pouvoir l’exécuter.

L’entraîneur, en plus de friser la crise d’apoplexie, fonctionne de manière totalement polymorphique. Il n’aura pas tout perdu. Lors de l’exécution du code, tous les joueurs qui recevront le message seront de type `Attaquant`, `Gardien` ou `Defenseur`. Il semble donc que les objets du tableau joueur, en fait tous les joueurs sur le terrain, peuvent bénéficier de deux typages, un typage dit statique, celui que seul le compilateur comprend et vérifie (le seul connu de l’entraîneur), et un typage dynamique, qui se révélera seulement à l’exécution.

En C++

```
class Entraineur {
 private:
  Joueur* lesJoueurs[];
 public:
  Entraineur(Joueur* lesJoueurs[]) {
   for (int i=0; i<3; i++)
    this->lesJoueurs[i] = lesJoueurs[i];
  }
  void panique() {
```

```
    cout << "C'est la panique" << endl;
    for (int i=0; i<3; i++)
     lesJoueurs[i] ->avance(); // le même message à tous les joueurs - attention !!! polymorphisme
   }
};
```

Rien de très différent par rapport à la version Java, à ceci près que les joueurs apparaissent dans un tableau de pointeurs, ce qui nous oblige à assigner les pointeurs, un à un, à l'aide d'une boucle. Les tableaux en C++ ne sont pas des objets, et il est nécessaire de les manipuler élément par élément.

En C#

```
class Entraineur {
 private Joueur[] lesJoueurs;
 public Entraineur(Joueur[] lesJoueurs) {
  this.lesJoueurs = lesJoueurs;
 }
 public void panique() {
  Console.WriteLine("C'est la panique");
  for (int i=0; i<lesJoueurs.Length; i++)
   lesJoueurs[i].avance(); /* le même message à tous les joueurs - attention !!! polymorphisme */
 }
}
```

Exactement le même entraîneur qu'en Java.

En Python

```
class Entraineur:
        __lesJoueurs={}
        def __init__(self,lesJoueurs):
               self.__lesJoueurs=lesJoueurs
        def panique(self):
               print "C'est la panique"
               i=0
               while i<len(self.__lesJoueurs):
                      self.__lesJoueurs[i].avance()
                      i+=1
```

En PHP 5

```
class Entraineur {
      private $lesJoueurs;

      public function __construct($lesJoueurs){
            $this->lesJoueurs = $lesJoueurs;
      }

      public function panique() {
            print ("C'est la panique <br> \n");
            for ($i=0; $i<3; $i++) {
                  $this->lesJoueurs[$i]->avance();
            }
      }
}
```

En Python et en PHP 5, on retrouve le même entraîneur (c’est le sort des bons entraîneurs de devoir se partager à ce point) qu’en Java, en C# et en C++ à quelques détails syntaxiques insignifiants près.

Passons maintenant au bouquet final

… c’est-à-dire la méthode principale, afin, qu’en plus des joueurs, les Romains en viennent à s’empoigner.

En Java

```
public class Football {
 public static void main(String[] args) {
  Balle uneBalle   = new Balle();       // création de la balle
  Joueur lesJoueurs[] = new Joueur[3];      // création de l'objet tableau
  lesJoueurs[0]    = new Gardien(uneBalle);  // création du premier joueur, un gardien
  lesJoueurs[1]    = new Defenseur(uneBalle); // création du deuxième joueur, un défenseur
  lesJoueurs[2]    = new Attaquant(uneBalle); // création du troisième joueur, un attaquant

  Entraineur unEntraineur = new Entraineur(lesJoueurs); // création de l'entraîneur

  System.out.println("******* d'abord les joueurs *****");
  for (int i=0; i<lesJoueurs.length; i++)
   lesJoueurs[i].interagitBalle();

  System.out.println("******* puis l'entraineur *****");
  for (int i=0; i<6; i++)
   unEntraineur.panique();
 }
}
```

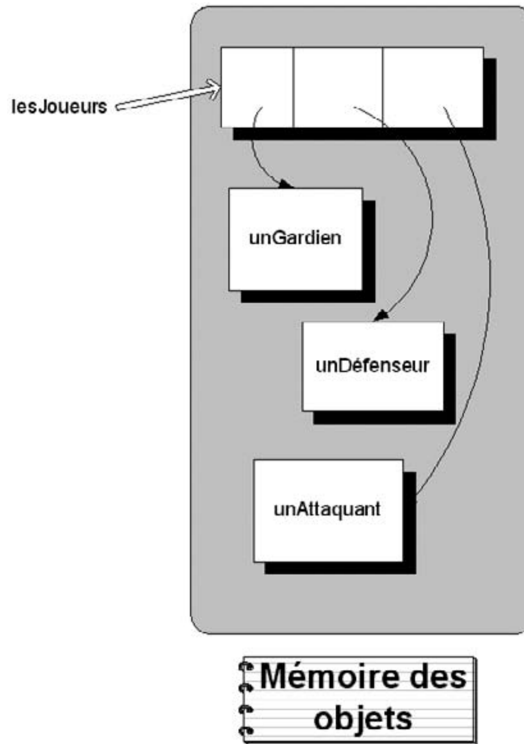
Dans l’ordre, on crée d’abord l’objet `Balle`, puis un tableau de 3 joueurs (on se limitera pour des raisons évidentes à 3, mais à 11 ce serait pareil). À ce stade-ci, le tableau des joueurs est typé `Joueur`. En fait, on construit un tableau de référents, chacun des référents étant typé statiquement comme Joueur.

Ensuite, on crée trois joueurs de type différent : un gardien, un défenseur et un attaquant. Comme la figure ci-après l’illustre, quatre objets sont stockés en mémoire, un pour le tableau de référents et trois pour les joueurs. Au moment de l’exécution, les objets finaux, référés par les trois éléments du tableau, ne sont plus du type de la superclasse `Joueur`, mais chacun d’une sous-classe différente. Il faut se souvenir que l’opération `new` ne s’effectue que pendant l’exécution. Il n’est donc pas possible de prévoir, avant l’exécution, au moment de la compilation, de quel type dynamique sera l’objet. Nous pourrions très facilement nous retrouver dans une situation dans laquelle la création de l’objet serait conditionnée par une information à découvrir pendant l’exécution. Cela veut dire, qu’avant l’exécution, on ne peut présager avec certitude de la classe finale dont l’objet sera une instance. La seule garantie que l’on ait est le typage statique de cet objet, qui est forcément une superclasse de la classe finale.

Dans de nombreux cas, la classe qui déclare l’objet et la classe qui suit l’opération `new` sont les mêmes, comme lorsque nous écrivons : `O1 unObjetO1 = new O1().`

Mais, ici, la donne a changé. Nous nous retrouvons dans une nouvelle situation, assez singulière, où la classe à gauche et à droite de cette instruction peuvent être différentes (mais pas indépendantes). La classe à droite,

Figure 12-4
Les 4 objets nécessaires au stockage de l'objet tableau de joueurs et des 3 objets joueurs



c'est-à-dire, la classe finale, révélée au moment de l'exécution, se doit d'être absolument ou la même ou une sous-classe de la classe à gauche, c'est-à-dire la classe fournie par le typage statique. On peut écrire, sans heurter le compilateur : `O1 unObjetO1 = new FilsO1()`.

En substance, le compilateur se satisfait, pour la justesse syntaxique, d'une superclasse, alors que le type final, à l'exécution, pourrait être une sous-classe de celle-ci. Rien de choquant à cela, étant donné le principe de substitution, qui nous dit que, si la superclasse peut le faire, toute sous-classe le fera également sans problème. Mais nous devrons, à partir de maintenant, nous efforcer de différencier le type statique, la classe à gauche, la seule importante pour le compilateur, du type dynamique, la classe à droite, la seule vraiment importante pour l'exécution du programme. Cette différenciation sera capitale pour comprendre le fonctionnement particulier et distinct des trois langages de programmation, C++, C# et Java, pour qui le typage explicite compte vraiment. En l'absence de compilateur, la situation est foncièrement différente pour Python et PHP 5.

Un même ordre mais une exécution différente

Dans la suite de la méthode `main` de Java, on crée un objet entraîneur. Finalement, on envoie le même message `interagitBalle()` aux trois joueurs et le message `panique()` à l'entraîneur, en sachant que l'exécution de ce message par l'entraîneur, aura, à son tour, comme effet d'envoyer le message `avance()` aux trois joueurs. L'entraîneur hurle ce message 6 fois de suite.

Lançons la simulation et affichons le résultat :

Résultat

```
********************      Résultat : ************************
******* d'abord les joueurs *****
Je tape la balle avec le pied
la balle bouge
Je prends la balle avec les mains
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
******* puis l'entraineur *****
C'est la panique
Moi gardien, je peux encore prendre la balle avec les mains
la position actuelle du Gardien est 0
la position actuelle du Defenseur est 20
la position actuelle du Attaquant est 100
C'est la panique
la position actuelle du Defenseur est 40
la position actuelle du Attaquant est 120
C'est la panique
la position actuelle du Defenseur est 60
la position actuelle du Attaquant est 140
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Defenseur est 80
la position actuelle du Attaquant est 160
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Attaquant est 180
moi attaquant je fais attention au hors-jeu
C'est la panique
*********************************************************************
```

D'abord, le même message `interagitBalle()` est lancé aux trois joueurs. On s'aperçoit que ce message est, de fait, exécuté différemment selon le type de joueur. Le défenseur et l'attaquant exécutent celui défini par défaut pour tous les joueurs, alors que le gardien, lui, exécute sa version particulière, celle qu'il a redéfinie. Cela reflète bien le mode de découverte ascendant que Java met en œuvre pour découvrir la méthode à exécuter. Une fois le type de l'objet identifié lors de l'exécution, la méthode à exécuter sera d'abord recherchée dans la zone mémoire allouée à ce type.

Si la méthode n'est pas trouvée, on « grimpera », en quête de celle-ci, dans les zones mémoires allouées aux superclasses. On dit de Java qu'il est un langage polymorphique par défaut, c'est-à-dire, qu'à défaut d'autre chose, il donne toujours priorité, dans le choix de la méthode à exécuter, à celle qui correspond au type dynamique. Aussi, si le type dynamique est donc une sous-classe du type statique ; le compilateur aura préalablement vérifié la cohérence et la justesse de la démarche. À l'exécution, le choix de la méthode adéquate se fera sur l'instant, après un processus de recherche, qui, comme souvent en OO, ralentit le processus d'exécution, mais de manière acceptable. Chaque objet possède en fait un attribut supplémentaire mais caché, son type dynamique. Lorsqu'il reçoit un ordre d'exécution de méthode, il « s'introspecte », découvre sa classe définitive et s'assure que la méthode exécutée est bien celle déclarée dans cette classe-là.

Ensuite, l'entraîneur envoie le même message `avance()` aux trois joueurs. Ici, également, le message sera exécuté de trois manières différentes. Chaque joueur, grâce à la présence de la méthode `toString()`, nous informera sur la nature de sa classe et avancera de 20, si sa méthode le lui permet. On constate qu'au fur et à mesure, de moins en moins de joueurs pourront avancer, car tous arriveront à la limite de leur déplacement. Le rôle de la méthode `toString()` est de révéler, une fois de plus, le mécanisme polymorphique qui permet la participation des différentes sous-classes, bien que le tableau de joueurs reste typé d'une seule et même superclasse. Passons maintenant au C++, et apprêtons-nous à découvrir un comportement plutôt surprenant.

C++ : un comportement surprenant

```
int main(int argc, char* argv[]){
 Balle uneBalle;
 Joueur* lesJoueurs[3];
 lesJoueurs[0] = new Gardien(&uneBalle);
 lesJoueurs[1] = new Defenseur(&uneBalle);
 lesJoueurs[2] = new Attaquant(&uneBalle);
 cout <<"******* d'abord les joueurs *****" << endl;
 for (int i=0; i<3; i++)
  lesJoueurs[i]->interagitBalle();
 Entraineur unEntraineur(lesJoueurs);
 cout << "******* puis l'entraineur *****" << endl;
 for (int j=0; j<6; j++)
  unEntraineur.panique();
 return 0;
}
```

La fonction `main()` n'a rien de très particulier. Dans un premier temps, nous laisserons le mot-clé `virtual`, présent dans la déclaration des méthodes `interagitBalle()` et `avance()`, en commentaire, c'est-à-dire désactivé. Dans sa syntaxe, la version de code qui en résulte est la plus proche du code Java que nous venons d'exécuter. Pourtant, voici le résultat obtenu :

Résultat

```
Résultat de l'exécution du C++ sans déclarer les méthodes comme « virtual » :
******* d'abord les joueurs *****
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
******* puis l'entraineur *****
C'est la panique
la position actuelle du class Joueur joueur est 0
la position actuelle du class Joueur joueur est 20
la position actuelle du class Joueur joueur est 100
C'est la panique
la position actuelle du class Joueur joueur est 20
la position actuelle du class Joueur joueur est 40
la position actuelle du class Joueur joueur est 120
C'est la panique
```

```
la position actuelle du class Joueur joueur est 40
la position actuelle du class Joueur joueur est 60
la position actuelle du class Joueur joueur est 140
C'est la panique
la position actuelle du class Joueur joueur est 60
la position actuelle du class Joueur joueur est 80
la position actuelle du class Joueur joueur est 160
C'est la panique
la position actuelle du class Joueur joueur est 80
la position actuelle du class Joueur joueur est 100
la position actuelle du class Joueur joueur est 180
C'est la panique
la position actuelle du class Joueur joueur est 100
la position actuelle du class Joueur joueur est 120
la position actuelle du class Joueur joueur est 200
******************************************************************
```

Que ce soit lors de l’exécution du message `interagitBalle()` sur les trois joueurs ou du message `avance()`, nous constatons qu’au contraire de Java, en C++, le type statique prime sur le type dynamique. Par exemple, la méthode `avance()` s’exécutera sans limitation, c’est-à-dire dans sa version par défaut, octroyant le droit au gardien de faire un pas de deux dans le rectangle adverse, et aux attaquants de se noyer dans la foule. On peut lire dans le résultat que tous les joueurs se comportent, en effet, comme des `Joueurs`, et non dans leur version plus spécifique.

Indépendamment du type dynamique, c’est-à-dire de la sous-classe, celle qui caractérise vraiment les joueurs en définitive, le compilateur a le dernier mot et force le type statique au détriment du type dynamique, y compris lors de l’exécution. C’est plutôt déconcertant, car cela ne correspond pas du tout au comportement naturel que l’on serait en droit d’attendre. À quoi bon redéfinir des méthodes dans les sous-classes, si celles-ci n’ont pas la primeur lors du déclenchement du message qui les concerne ? En fait, encore une fois, C++ favorise l’optimisation sur la cohérence sémantique. La raison est à rechercher, en partie, toujours dans ce lourd tribut payé au C, langage procédural par excellence. Il est plus optimal de faire le lien entre la méthode et l’objet lors de l’étape de compilation que lors de l’étape d’exécution.

Aucune recherche de méthode, ralentissant l’exécution du programme, ne sera plus nécessaire pendant l’exécution. On dit de C++ qu’il n’est pas un langage polymorphique par défaut, comme l’était historiquement Smalltalk, et comme devrait l’être, là encore selon la charte de l’OO, tous les langages OO dignes de cette étiquette. Encore une fois, également, C++ décide que vous êtes adultes et vaccinés, et que c’est à vous de faire le choix entre l’optimisation ou la cohérence sémantique. Vous voulez du polymorphisme, les performances en temps calcul risquent d’en prendre un coup, mais, qu’à cela ne tienne, il suffit de retirer les commentaires qui entourent le mot-clé `virtual` dans la déclaration des deux méthodes redéfinies. En rendant les deux méthodes « virtuelles », voilà le nouveau résultat obtenu par le code C++ parfaitement en phase avec le résultat obtenu précédemment par Java.

Nouveau résultat C++ en déclarant les deux méthodes virtuelles :

```
******* d'abord les joueurs *****
Je tape la balle avec le pied
la balle bouge
Je prends la balle avec les mains
Je tape la balle avec le pied
la balle bouge
```

```
Je tape la balle avec le pied
la balle bouge
******* puis l'entraineur *****
C'est la panique
Moi gardien, je peux encore prendre la balle avec les mains
la position actuelle du class Gardien joueur est 0
la position actuelle du class Defenseur joueur est 20
la position actuelle du class Attaquant joueur est 100
C'est la panique
la position actuelle du class Defenseur joueur est 40
la position actuelle du class Attaquant joueur est 120
C'est la panique
la position actuelle du class Defenseur joueur est 60
la position actuelle du class Attaquant joueur est 140
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du class Defenseur joueur est 80
la position actuelle du class Attaquant joueur est 160
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du class Attaquant joueur est 180
moi attaquant je fais attention au hors-jeu
C'est la panique
```

Moyennant la présence du mot-clé `virtual` dans la déclaration des méthodes, C++ se comporte, d'un point de vue polymorphique, comme Java. Pour rendre le polymorphisme possible, il faut que la liaison entre l'objet et la méthode qui s'exécutera sur lui soit établie pendant l'exécution, le compilateur s'étant simplement assuré de la possibilité de la chose. Pour que cette liaison s'effectue, il faut, comme indiqué dans la figure ci-après, que l'objet, parmi ses attributs, en possède un supplémentaire, caché au programmeur, qui contienne l'information sur la zone mémoire où se situe la méthode.

En Java, c'est d'office le cas. En C++, ce sera le cas dès qu'une méthode de la classe est déclarée comme virtuelle. Un pointeur additionnel sera nécessaire par objet, pointant vers une table additionnelle par classe, indiquant pour chaque méthode virtuelle où se trouve la bonne implémentation à exécuter. En C++, la simple déclaration d'une méthode virtuelle provoque de ce fait un accroissement de mémoire, alloué pour les objets et pour la table, et un ralentissement résultant de la découverte de la méthode appropriée à l'aide des pointeurs présents dans la table. C'est pour cela que C++ donne la possibilité, au détriment de la simplicité et du comportement intuitif qui en résulte, de contourner le polymorphisme. Il favorise le temps calcul au détriment d'une certaine logique comportementale.

Polymorphisme : uniquement possible dans la mémoire tas

Le polymorphisme, à la base, permet qu'un même objet soit typé statiquement et dynamiquement de manière différente. Si cela est parfaitement possible avec les objets stockés dynamiquement dans la mémoire tas, cela est beaucoup plus délicat avec les objets stockés statiquement dans la mémoire pile. En C++, la simple instruction `O1 o1` crée l'objet `o1`. Pour modifier l'affectation dynamiquement, il faudrait dans le cours du programme pouvoir écrire : `o1 = filsO1`, avec l'objet `filsO1` instance de `FilsO1` sous-classe de `O1`.

Avec le principe de substitution, l'écriture est possible, mais le résultat est partiellement satisfaisant, car il faut que la zone mémoire initialement prévue pour recevoir `o1` puisse maintenant contenir `filsO1`.

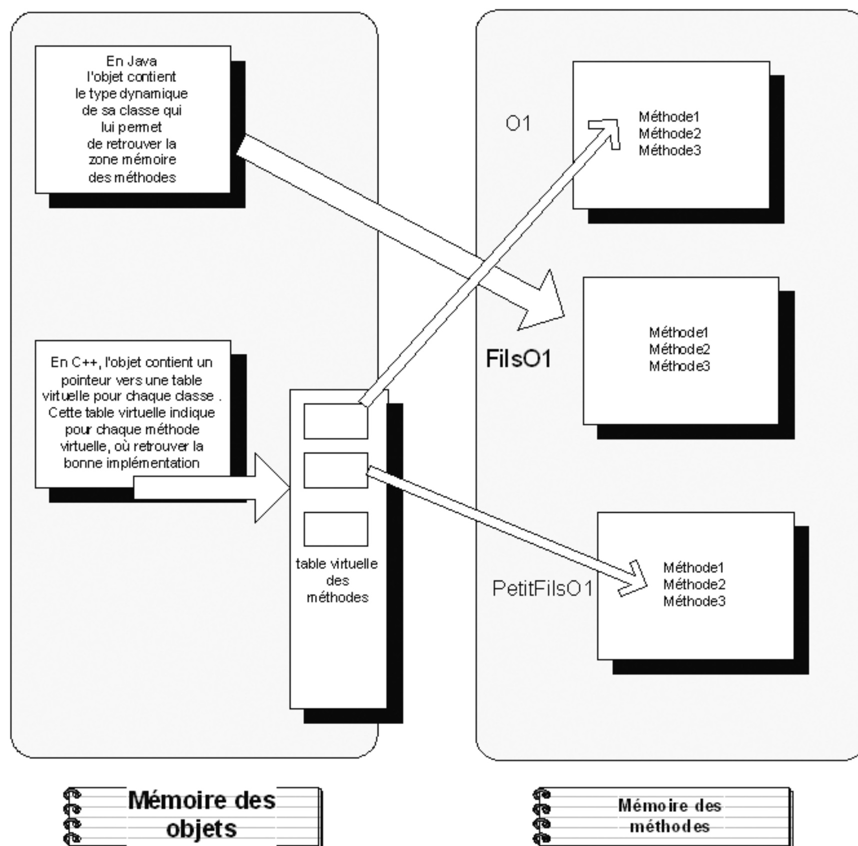


Figure 12-5
La mise en mémoire de la pratique du polymorphisme.

Quand on sait qu'il est fréquent que les objets des sous-classes soient plus volumineux que les objets de la superclasse, on ne s'étonnera pas que cette affectation d'un objet d'une sous-classe à la place de celui d'une superclasse ait un prix : la perte des attributs propres à la sous-classe et surtout la perte du pointeur vers les méthodes virtuelles. Aucun typage dynamique n'est, de ce fait, possible pour des objets stockés sur la pile (cela revient toujours à une forme de typage statique, prédéterminé par le compilateur), et ils ne peuvent en aucun cas bénéficier du polymorphisme qui exige la manipulation de référents. En présence des référents, l'instruction `o1 = filsO1` n'a comme seul effet que de faire pointer le référent `o1` vers l'objet précédemment référé par `filsO1`.

En C#

```
public class Football{
 public static void Main(){
  Balle uneBalle         = new Balle();
  Joueur[] lesJoueurs    = new Joueur[3];
  lesJoueurs[0]          = new Gardien(uneBalle);
  lesJoueurs[1]          = new Defenseur(uneBalle);
  lesJoueurs[2]          = new Attaquant(uneBalle);
```

```
    Entraineur unEntraineur = new Entraineur(lesJoueurs);
    Console.WriteLine("******* d'abord les joueurs *****");
    for (int i = 0; i<lesJoueurs.Length; i++)
     lesJoueurs[i].interagitBalle();
    Console.WriteLine("******* puis l'entraineur *****");
    for (int i = 0; i<6; i++)
     unEntraineur.panique();
   }
  }
```

On ne note rien de particulier dans l'écriture de la classe principale, qui ressemble à s'y méprendre à du Java. Cependant, du point de vue polymorphique, C# se situe entre les deux langages précédents. C'est l'avantage d'être le troisième, et c'est pour cela que, précisément, nous le traitons en troisième lieu. Si on laisse le programme tourner comme montré dans le code jusqu'à présent, c'est-à-dire, sans déclarer les méthodes à redéfinir `virtual`, il est alors obligatoire de rajouter le mot-clé `new` lors de la redéfinition des méthodes. En présence de ce mot-clé, le résultat est non polymorphique comme vous le constatez :

Résultat

Résultat du C# sans `virtual/override` mais en présence de `new` :

```
******* d'abord les joueurs *****
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
******* puis l'entraineur *****
C'est la panique
la position actuelle du Gardien est 0
la position actuelle du Defenseur est 20
la position actuelle du Attaquant est 100
C'est la panique
la position actuelle du Gardien est 10
la position actuelle du Defenseur est 30
la position actuelle du Attaquant est 110
C'est la panique
la position actuelle du Gardien est 20
la position actuelle du Defenseur est 40
la position actuelle du Attaquant est 120
C'est la panique
la position actuelle du Gardien est 30
la position actuelle du Defenseur est 50
la position actuelle du Attaquant est 130
C'est la panique
la position actuelle du Gardien est 40
la position actuelle du Defenseur est 60
la position actuelle du Attaquant est 140
C'est la panique
la position actuelle du Gardien est 50
la position actuelle du Defenseur est 70
la position actuelle du Attaquant est 150
```

Résultat en déclarant les méthodes à re-définir `virtual` et les méthodes re-définies `override` :

```
******* d'abord les joueurs *****
Je tape la balle avec le pied
la balle bouge
Je prends la balle avec les mains
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
******* puis l'entraineur *****
C'est la panique
Moi gardien, je peux encore prendre la balle avec les mains
la position actuelle du Gardien est 0
la position actuelle du Defenseur est 20
la position actuelle du Attaquant est 100
C'est la panique
la position actuelle du Defenseur est 40
la position actuelle du Attaquant est 120
C'est la panique
la position actuelle du Defenseur est 60
la position actuelle du Attaquant est 140
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Defenseur est 80
la position actuelle du Attaquant est 160
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Attaquant est 180
moi attaquant je fais attention au hors-jeu
C'est la panique
```

En revanche, en présence du couple `virtual/override`, on obtient bien le résultat polymorphique attendu. En fait, C# coupe vraiment la poire en deux, ou opte pour un jugement de Salomon. Il considère, d'abord, qu'il n'y a plus de comportement par défaut mais, à la place, qu'il y a deux comportements possibles. Les deux sont tout aussi adoptables, l'un met l'accent sur les performances l'autre sur une certaine logique comportementale, en optant pour une déclaration particulière des méthodes concernées. L'absence de comportement obtenu « gratuitement » ou par défaut contraint à maîtriser parfaitement ce que vous faites et les choix possibles. Enfin, tout cela ne concerne que les classes en C# et nullement les structures (comme les objets présents sur la pile dans le cas du C++), puisque celles-ci ne peuvent hériter entre elles. L'addition du `new` force la main, marque le coup, et indique explicitement que la redéfinition de cette méthode dans la sous-classe, ne se verra utilisée qu'en présence d'un objet typé statiquement par cette sous-classe.

En Python

```
uneBalle=Balle()
lesJoueurs={}
lesJoueurs[0]=Gardien(uneBalle)
lesJoueurs[1]=Defenseur(uneBalle)
lesJoueurs[2]=Attaquant(uneBalle)
unEntraineur=Entraineur(lesJoueurs)
```

```
print "****** d'abord les joueurs ******"
i=0
while i<len(lesJoueurs):
     lesJoueurs[i].interagitBalle()
     i+=1
print "****** puis l'entraineur ******"
i=0
while i<6:
     unEntraineur.panique()
     i+=1
```

Résultats

```
****** d'abord les joueurs ******
Je tape la balle avec le pied
la balle bouge
Je prends la Balle avec les mains
Je tape la balle avec le pied
la balle bouge
Je tape la balle avec le pied
la balle bouge
****** puis l'entraineur ******
C'est la panique
Moi gardien, je peux encore prendre la balle avec les mains
la position actuelle du Gardien est 0
la position actuelle du Defenseur est 20
la position actuelle du Attaquant est 100
C'est la panique
la position actuelle du Defenseur est 40
la position actuelle du Attaquant est 120
C'est la panique
la position actuelle du Defenseur est 60
la position actuelle du Attaquant est 140
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Defenseur est 80
la position actuelle du Attaquant est 160
moi attaquant je fais attention au hors-jeu
C'est la panique
la position actuelle du Attaquant est 180
moi attaquant je fais attention au hors-jeu
C'est la panique
moi attaquant je fais attention au hors-jeu
```

En PHP 5

```
$uneBalle = new Balle();
$lesJoueurs[0] = new Gardien($uneBalle);
$lesJoueurs[1] = new Defenseur($uneBalle);
$lesJoueurs[2] = new Attaquant($uneBalle);
```

```
$unEntraineur = new Entraineur($lesJoueurs);
print ("********* d'abord les joueurs ******* <br> \n");
for ($i = 0; $i<3; $i++) {
    $lesJoueurs[$i]->interagitBalle();
  }
 print ("******** puis l'entraineur ******** <br> \n");
 for ($i = 0; $i<6; $i++) {
       $unEntraineur->panique();
  }         i+=1
```

Agréable surprise enfin pour Python et PHP 5. Ils se conforment bien tous deux à la charte du bon langage OO car, à l’instar de Java, il sont polymorphiques par défaut. Sans rien ajouter pour ce faire, la classe dynamique prime sur la classe statique lors de l’exécution du code. Remarquez toutefois que cela ne leur pose pas trop de problème, puisque ces langages ont purement et simplement supprimé le typage statique, qui est le seul vérifié par le compilateur. C’est bien lors de la réception du message que l’objet vérifiera quelle version de celui-ci il doit exécuter, mais cela ne pose aucun problème, vu que nul compilateur ne les aura préalablement aiguillé sur une mauvaise piste.

Polymorphisme possible mais différent dans les cinq langages

La mise en place du polymorphisme différencie les cinq langages de programmation de manière sensible. C++ se comporte, par défaut, de manière non polymorphique, Java, Python et PHP 5 font le contraire, et C# considère qu’il n’y a plus lieu de laisser une version par défaut mais de préciser ce que vous cherchez à faire.

Quand la sous-classe doit se démarquer pour marquer

Rajoutons dans notre simulation Java du match de football, et dans la sous-classe `Attaquant`, la méthode suivante : `marqueUnBut()`, comme indiqué ci-après :

```
class Attaquant extends Joueur {
 public Attaquant (Balle laBalle) {
  super(laBalle);
  setPosition(100);
 }
 public void avance() {
  if (getPosition() < 200) {
   super.avance();
   if (getPosition() > 150)
    System.out.println("moi attaquant je fais attention au hors-jeu");
  }
 }
 public void marqueUnBut() {
  System.out.println("youpiiiii..... j'ai marqué... !!");
 }
}
```

On se place dans le cas extrême, où seuls les attaquants sont autorisés à marquer. Il serait somme toute assez naturel de les en autoriser. Or, le compilateur, et ce en dépit des cris désespérés de l’entraîneur, fait une totale obstruction. Si dans la méthode `main`, vous écrivez :

```
lesJoueurs[2].marqueLeBut() ;
```

Bien que tout leur permette de le faire, car ils sont bien attaquants et peuvent marquer des buts, cette instruction générera une erreur de compilation. C'est normal, puisque le rôle premier du compilateur est de vérifier que tout envoi de message est conforme au typage statique de l'objet. Nous avons bien dit au typage statique et non au typage dynamique, puisque ce type est supposé non connu au moment de la compilation. Vous pourriez vous étonner de l'étroitesse de vue du compilateur. Dans l'instruction :

```
lesJoueurs[2] = new Attaquant(uneBalle);
```

ce même compilateur pourrait se rendre compte que le type final de l'objet, le type à l'exécution, le seul qui compte *in fine*, est `Attaquant` et, donc, que `lesJoueurs[2]` peuvent, de fait, marquer un but. Dans un cas semblable, vous avez tout à fait raison, il le pourrait.

Mais considérons maintenant un cas plus général, correspondant au petit code suivant :

```
int a;
Joueur unJoueur;
readConsole( a ); /* on imagine une instruction qui permet de donner au
                   * clavier la valeur de a alors que le code s'exécute,
                   * et qui existe dans tous les langages de programmation */
if (a > 1)
 unJoueur = new Gardien() ;
else
 unJoueur = new Attaquant() ;
unJoueur.marqueUnBut() ;
```

Ici, vous admettrez que, si le compilateur se basait sur le type dynamique, il serait bien en peine de savoir si la réception du message par le joueur est possible ou pas. Et c'est bien pour cela que le compilateur, féroce, mais néanmoins prudent, ne se base, pour sa vérification de la conformité des envois de message, que sur le typage statique.

Les attaquants participent à un « casting »

D'où un problème basique, comment détourner l'attention du compilateur ? Comment lui faire comprendre que, bien que `leMarquageDeBut` ne soit pas vrai de tous les joueurs, nous savons, nous, programmeurs compétents, que le `joueur[2]` est bien un attaquant et qu'il peut se le permettre. La solution est de recourir au « casting », traduit de différentes manières en français : « transtypage », « coercion » (on en passe et des meilleures), et qui consiste à forcer la main au compilateur de la manière suivante :

```
((Attaquant)lesJoueurs[2]).marqueUnBut()
```

Cela revient à dire ceci. On sait qu'il n'est pas prévu que tous les joueurs marquent des buts, mais on sait également quelque chose que toi, compilateur, tu ne peux pas savoir (car cette information sera obtenue seulement pendant l'exécution) : le deuxième joueur du tableau est bien un attaquant, et, en tant que tel, il peut marquer un but. Le compilateur acceptera un `casting` d'une classe dans une de ses sous-classes, mais dans aucune autre. Il est évident qu'il n'y aura jamais lieu d'opérer ce `casting` dans le sens contraire. Le principe de substitution nous permet toujours de faire passer une sous-classe pour sa superclasse (on parle alors de « casting implicite »). Cela, c'est complètement admis et parfaitement normal. Ce qui ne l'est plus, c'est de faire passer la superclasse pour sa sous-classe. Car, en effet, rien ne nous incite à penser que cela puisse fonctionner (rapellez-vous dans le chapitre précédent de la Mazda que l'on traiterait comme une Ferrari…

Et, de fait, cela pourrait ne pas marcher. Comme à chaque fois que vous désactivez un système de protection, cela peut se retourner contre vous. Supposons qu’alors que notre programme compile merveilleusement, un gardien plutôt qu’un attaquant soit installé à la place du `joueur[2]`, comme indiqué ci-après :

```
lesJoueurs[2] = new Gardien(); /* nous avons maintenant un gardien en place d’un attaquant.*/
```

Le compilateur ne tiquera pas quand il lira l’instruction : `((Attaquant)lesJoueurs[2]).marqueUnBut()` puisqu’il ne connaît pas le type final du `joueur[2]`. Mais, lors de l’exécution, une erreur surviendra, de type « mauvais casting », comme montré ci-après, lorsque le code Java s’exécute :

```
C'est la panique
la position actuelle du Defenseur est 80
C'est la panique
C'est la panique
java.lang.ClassCastException: Gardien
 at Football.main(Football.java:162)
Exception in thread "main"
```

Éviter les « mauvais castings »

Une erreur de type « mauvais casting » apparaît. Le programme s’attendait à recevoir un gardien pendant l’exécution, et vous lui passez un attaquant à la place. Comme cette erreur se produit pendant l’exécution, et qu’il vaut mieux tenter de prévenir toute forme d’erreur dès l’écriture du code, il y a deux manières de procéder. Vous pourriez accepter l’erreur si elle survient, et recourir alors à une gestion d’exception que Java encourage toujours dans l’écriture du code. (il vous faudrait alors faire une gestion d’exception `ClassCastException` pour tout `casting`). Mais il y a mieux à faire : empêcher une telle erreur de se produire. Vous pouvez, à l’aide de l’opérateur `instanceof` qui, en Java (il existe le même en PHP 5), renvoie le type dynamique de l’objet, vérifier que vous opérez bien un « casting » possible.

L’écriture devient alors :

```
 if (lesJoueurs[2] instanceof Attaquant)
  ((Attaquant)lesJoueurs[2]).marqueUnBut();
```

En fait, il vous revient de forcer pendant l’exécution la vérification du type, avant d’opérer le `casting`. Cela ne change évidemment rien à la compilation, mais cela permet de n’envoyer le message à l’objet que si celui-ci est apte à le recevoir et ainsi d’éviter que l’exécution ne se « plante ».

Bien que les précautions à prendre soient les mêmes, et dans le même esprit, la manière de procéder se transforme légèrement en C++ et en C#. Ils autorisent également le `casting`, mais encourageraient et feraient la vérification de type plutôt de la manière suivante :

En C++

```
class Attaquant : public Joueur {
 public:
  Attaquant (Balle* laBalle) : Joueur(laBalle) {
   setPosition(100);
  }
```

```
    void avance() {
     if (getPosition() < 200) {
       Joueur::avance();
       if (getPosition() > 150)
       {
        cout<<"moi attaquant je fais attention au hors-jeu"<<endl;
       }
     }
    }
    void marqueUnBut() {
     cout << "youpiiii.... j'ai marqué..... " << endl;
    }
};
int main(int argc, char* argv[]) {
 Balle uneBalle;
 Joueur* lesJoueurs[3];
 lesJoueurs[0] = new Gardien(&uneBalle);
 lesJoueurs[1] = new Defenseur(&uneBalle);
 lesJoueurs[2] = new Attaquant(&uneBalle);

 cout << "******* d'abord les joueurs *****"<< endl;
 for (int i=0; i<3; i++)
  lesJoueurs[i]->interagitBalle();
 Entraineur unEntraineur(lesJoueurs);

 cout << "******* puis l'entraineur *****" << endl;
 for (int j=0; j<6; j++)
  unEntraineur.panique();
 Attaquant *unAttaquant = dynamic_cast<Attaquant*>(lesJoueurs[2]) ; /* afin de vérifier le type
➥dynamique de l'objet */

 if (unAttaquant != 0)
  unAttaquant->marqueUnBut();
 return 0;
}
```

En C#

```
 Attaquant unAttaquant = lesJoueurs[2] as Attaquant;
 if (unAttaquant!= null)
  unAttaquant.marqueUnBut();
```

En C# comme en C++, on force le `casting`. Ça passe ou ça « classe ». Si cela marche, c'est-à-dire si, à l'exécution, l'objet est bien du type dynamique de la classe dans laquelle on désire le « caster », le nouveau référent recevra la bonne adresse, sinon il recevra 0 ou null, mais l'envoi de message ne s'effectuera pas, bien heureusement.

Les problèmes de casting de ce type ne concernent pas Python et PHP 5, étant donné leur absence de typage des attributs ou des référents. De fait, dans ces langages, aucun compilateur ne vérifie préalablement la syntaxe et les types statiques. C'est seulement au moment de l'exécution que l'on découvrira tout ce qu'il y a à découvrir sur le type des objets auxquels sont destinés les messages. Rien de préalable n'entravera le cours d'exécution des messages. Si vous observez les codes Python et PHP 5, jamais le vecteur des joueurs n'a dû être typé, comme pour les trois langages précédents, par la classe `Joueur`, et ce parce que la liste ou la collection peuvent exister sans typage statique.

Python et PHP 5 : tout se passe à l'exécution

Python et PHP 5 s'interprétant, c'est-à-dire exécutant les instructions du code, au fur et à mesure de leur rencontre, la traduction en langage exécutable s'effectue « en ligne ». Il est suivi directement de l'exécution, laissant donc à cette phase d'exécution le soin de découvrir des erreurs qui, dans d'autres langages OO, seraient découvertes lors de la compilation. Tous les « viols » et les incohérences de typage, par exemple, l'envoi d'un message à un objet qui n'est pas destiné par sa classe à recevoir ce message, se découvriront donc lors de l'exécution. Quand on sait le sang d'encre que se font C++, Java et C# pour prévenir ce type d'erreur par un typage fort et l'engagement à l'entrée du code d'un « videur-compilateur », employé à faire respecter à la lettre ce typage, on peut s'interroger sur cette option prise par ces deux langages. Leur défense s'appuie sur la simplicité et la rapidité de mise en œuvre pour aller directement à l'essentiel et ne se préoccuper que des fonctionalités premières du code. On peut donc les voir plus comme des langages de prototypage, susceptibles de céder la place, lors d'une phase plus « industrielle », à des langage plus contraignants, plus « safe » et plus rapide (surtout Python, PHP 5 restant un langage de prédilection pour les Maîtres du Web).

Le « casting » a mauvaise presse

Avouons-le tout de go, l'opération de `casting` a, en général, mauvaise réputation en programmation. D'ailleurs Stroustrup, inventeur du C++, regrette son omniprésence dans la programmation en Java ou C#. Cette manière de détourner l'attention du compilateur pour faire passer quelque chose pour ce que ce n'est pas a été largement fauteur de troubles dans des langages comme C et C++. En effet, l'utilisation malveillante ou simplement distraite du compilateur peut entraîner des effets plutôt brutaux et inélégants. Il a justement comme rôle de faire une vérification de la bonne utilisation des types, pourquoi délibérément lui fausser compagnie ?

Dans l'exemple décrit ci-dessus, il aurait été très facile d'éviter le casting en, comme le petit diagramme de classe l'illustre ci-dessous, rajoutant explicitement un référent de type `Attaquant` auprès de l'entraîneur, de manière à ce que ce dernier n'envoie qu'à ce seul `Attaquant` les seuls messages qui le concernent.

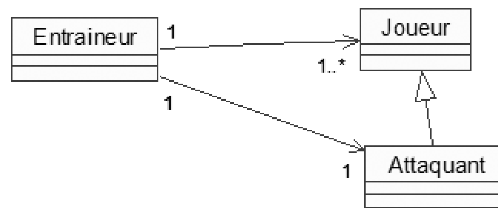


Figure 12-6

Diagramme de classe alternatif qui évitera à l'entraîneur de recourir au « casting » pour demander à son attaquant de marquer un but.

Cette solution est clairement celle du puriste, qui tente d'éviter par la compilation et le typage fort toute mauvaise surprise lors de l'exécution du code. Nous pensons néanmoins que, dans ce cas précis, et vu l'omniprésence de cette opération de « casting » en Java ou C#, la critique n'a plus exactement la même portée. D'abord, ce casting n'est toujours autorisé que dans certaines limites : une classe dans sa sous-classe, et rien d'autre. Ensuite, il survient souvent comme le juste pendant du polymorphisme. Or si le polymorphisme est, lui, très encouragé, il est difficile de protester contre une situation qui lui est souvent conséquente. Enfin, si son évitement prête à plus de contorsions étranges de la part du programmeur que sa simple acceptation, mais maîtrisée, il n'y a plus lieu d'hésiter.

Par exemple, sans recourir au référent `Attaquant` additionnel mentionné précédemment, une manière alternative de l'éviter serait, dans le cas du football, de déclarer la méthode `marqueUnBut()` chez tous les joueurs, mais de la vider de son contenu d'instructions, tant chez le gardien que le défenseur. Absurde non ? Faire comme si le gardien et le défenseur pouvaient marquer des buts alors qu'ils ne peuvent pas… L'unique consigne reste donc que le compilateur a toujours raison, mais que vous pouvez le forcer, de temps à autre, à relâcher son attention dans une partie du programme. Partie de programme que vous devez, en contrepartie, vous forcer de réaliser en redoublant d'attention, vu les possibles erreurs indésirables pendant l'exécution auxquelles vous exposez cette partie.

Polymorphisme et casting

Une conséquence du polymorphisme est le recours au « casting » qui vise à récupérer des fonctionnalités propres à l'une ou l'autre sous-classe lors de l'exécution d'un programme. Sa pratique est parfois délicate et demande une attention soutenue, car elle peut mener à des erreurs pendant la phase d'exécution. Java et C#, par exemple, par l'introduction de la généricité dans leurs dernières versions, tentent de diminuer ce recours. L'absence de typage statique dans Python et PHP 5 est bien évidemment une manière de contourner cette problématique, bien que cette absence ne les mette pas non plus à l'abri d'avatars ne survenant malheureusement qu'à l'exécution, lorsqu'on s'y attend le moins.

Redéfinition et encapsulation

Que se passe-t-il si, comme dans le petit code Java ci-dessous, la méthode que nous cherchons à redéfinir est déclarée `private` dans la superclasse.

```
class O1 {
      public void jeTravaillePourO1() {
            jeSuisPriveDansO1();
      }

      private /*protected*/ void jeSuisPriveDansO1() {
            System.out.println("je suis O1");
      }
}

public class FilsO1 extends O1 {
      public void jeSuisPriveDansO1() {
            System.out.println("je suis le Fils d'O1");
      }

      public static void main(String[] args) {
            O1 o1 = new FilsO1();
            o1.jeTravaillePourO1();
      }
}
```

Comme vous pouvez le voir à l'exécution, selon que vous déclariez la méthode `jeSuisPriveDansO1` de la superclasse `private` ou `protected`, c'est la version de la superclasse ou de la sous-classe qui s'exécutera. Or, si l'on s'en tient à la découverte des méthodes fonctionnant de manière ascendante en Java, ce devrait toujours être la version redéfinie qui s'exécute (donc celle de la sous-classe). Cependant, les langages OO considèrent à juste titre qu'une méthode déclarée comme `private` dans la superclasse, n'étant nullement accessible par la sous-classe (ce qui n'est pas le cas d'une méthode `protected`), ne peut se prêter à une quelconque redéfinition.

En fait, c’est comme si la méthode de la sous-classe était renommée implicitement par Java, afin d’éviter toute confusion possible. Cela n’est plus la même, ç’en est une autre ! C# vous évite ce genre de problème et de confusion en forçant les deux méthodes à posséder le même niveau d’accessibilité. De plus, vous ne pourrez jamais redéfinir une méthode déclarée `private` en C# (un bon point pour ce langage qui évite là une source patente de confusion).

Figure 12-7

Illustration de la ligue simulation de la Robocup.

Redéfinition de méthodes et multihéritage

Si deux classes redéfinissent toutes deux une méthode d’abord définie dans une superclasse qu’elles se partagent (et que toutes deux la redéfinissent en faisant appel à la méthode d’origine) et qu’à leur tour ces deux classes sont héritées par une seule classe (le type de situation problématique d’héritage en losange évoquée dans le chapitre précédent) et que cette dernière décide de redéfinir toujours la même méthode (en faisant également appel aux deux méthodes d’en haut) se pose, à nouveau, le problème de la présence une fois ou deux fois de la méthode de la superclasse du haut (le même problème que nous avons vu dans le chapitre précédent mais avec les méthodes et non plus les attributs de la classe au sommet). Tant Python que C++ offrent des solutions optionnelles pour faciliter le choix de l’implémentation.

Polymorphisme contre case-switch

On dit souvent que toute programmation qui exige qu'un objet, à la réception d'un message, teste sa nature intime par l'intermédiaire d'un `case-switch` ou d'une succession de `if-then` afin de savoir quelle methode exécuter, est une partie de code idéale pour réaliser un « polymorphisme ». Lorsque nous donnons cours de programmation OO, nous ne testons pas, au préalable les prérequis de chacun de nos élèves, de façon à adopter le cours pour chaucun. De même, chacun d'entre eux, à la réception du message que nous leur délivrons, ne s'interroge pas sur ses capacités propres afin de savoir comment digérer la matière. Une programmation polymorphique vous permet en effet d'éviter cette succesion de tests. Chaque sous-classe d'étudiant (n'y voyez rien de péjoratif) recevra ce cours à la manière de sa sous-classe. Un programme conçu de telle sorte évitera évidemment les réécritures qu'une série de tests ou un `case-switch` entraînerait suite au rajout d'une sous-classe.

La Robocup : Simulation League

Cette petite incursion logicielle du côté du football nous donne envie de vous parler de la Robocup. Cet événement annuel met en compétition des équipes de football constituées soit de robots, soit de joueurs programmés. Il existe plusieurs ligues : trois robotiques, selon la nature et la taille des robots ; cette année (2002), une quatrième devrait voir s'affronter les premiers robots humanoïdes, et une ligue de simulation. L'un des auteurs de cet ouvrage envoie ses étudiants y participer depuis trois années de suite. On peut dire qu'il les envoie au casse-pipe, car l'équipe se retrouve éliminée chaque année au premier tour, et ce par des scores défiant toute concurrence. Néanmoins, Coubertin faisant foi, cette expérience nous incite à encourager plus de monde encore à y participer, et surtout des étudiants, car il s'agit d'un excellent véhicule didactique de la programmation orientée objet.

En effet, vous aurez constaté que le football se prête merveilleusement à un développement de type OO, alors pourquoi ne pas y aller franco, et soumettre également une équipe. Nos étudiants passeront peut-être un tour l'année prochaine. L'idée d'un match de football entre robots germa dans la tête de chercheurs japonais (on aurait été étonné du contraire) dès 1993, mais il fallut attendre 1997 pour que la première compétition eut lieu à Nagoya au Japon. Depuis, cet événement se reproduit annuellement et réunit des milliers de participants. De nos jours, on peut considérer que 3000 chercheurs sont concernés de près ou de loin par la Robocup. Il est aujourd'hui, aussi paradoxal que celui puisse sembler au premier abord, beaucoup plus facile de concevoir, en informatique, un bon joueur d'échecs qu'un bon joueur de foot. La difficulté à reproduire nos facultés sensori-motrices y est pour beaucoup, bien évidemment, et pose d'extraordinaires défis pour les chercheurs en intelligence artificielle. Le constat majeur est qu'une intelligence désincarnée et découplée du monde environnant est bien plus à notre portée qu'une intelligence effective, utile et opérationnelle, à même ce monde. Le monde en simulation est autrement plus gérable que le monde réel, pour un être lui-même simulé. Par les nouveaux défis qu'elle pose : perception visuelle, motricité, temps réel, intelligence collective…, la Robocup est un irremplaçable moyen de promotion pour la recherche en robotique et en intelligence artificielle.

Le grand avantage de la ligue de simulation sur celle des ligues robotiques est qu'un tournevis n'est plus nécessaire, mais plutôt une bonne connaissance de la programmation OO, un bon sens commun, quelques manuels d'IA et ne pas souffrir du manque de sommeil et de régime pizza marguerita. Si l'objectif ultime est d'atteindre pour la moitié du XXIe siècle une équipe de robots humanoïdes capables de rivaliser avec des joueurs humains, la Robocup nous apparaît, pour notre part, comme un excellent exercice de programmation OO, en synergie avec les apports de l'intelligence artificielle.

Dans la version simulation, onze joueurs logiciels (le plus souvent programmés en C++) forment une équipe. Chacun de ces joueurs est un client du serveur, dont le programme décide de ce qu'il doit faire, et envoie le résultat de sa décision au serveur : accélérer, communiquer, frapper dans la balle, tourner la tête, tourner son corps. Chaque joueur possède une énergie décrémentée en fonction des actions entreprises, mais est capable également de doucement se régénérer. La puissance de ces actions dépend de cette énergie. Lorsque le serveur reçoit ces commandes, et afin de reproduire l'imprédictibilité inhérente au monde réel, il ajoute un bruit aléatoire aux mouvements des joueurs et de la balle, ainsi qu'aux décisions prises par les joueurs. La connexion client-serveur est de type UDP/IP. Les règles de la FIFA sont respectées autant que faire se peut : temps réglementaire (5 min par mi-temps), sortie de balle, hors-jeu, etc. Chaque joueur reçoit de la part du serveur des informations visuelles sur sa position (en fonction des objets qui l'entourent), l'imprécision de cette information s'accroissant avec la distance. Il perçoit également des communications provenant d'autres joueurs à sa portée, et des informations sur son état interne.

Bon match !

Exercices

Exercice 12.1

Réalisez le diagramme de classe UML, ainsi qu'une ébauche de code, dans un quelconque des trois langages de programmation, des classes suivantes. Une agence bancaire contient des comptes en banque de deux sortes : livret d'épargne et compte courant. Ce qui les différencie, c'est que, dans le premier compte, un retrait quelconque ne peut jamais rendre le solde négatif et que, dans le second, le retrait ne peut amener le solde en dessous de – 1000 euros. Une autre différence tient à la manière de calculer l'intérêt. Dans le premier cas, c'est la manière par défaut qui consiste à calculer l'intérêt multiplié par 2 % qui prévaut, dans le second, c'est la manière par défaut qui consiste à prendre la racine carrée. Utilisez l'appel des méthodes de la superclasse.

Exercice 12.2

Le « boot » par défaut d'un ordinateur consiste à charger le système d'exploitation présent sur le disque dur dans la mémoire RAM. On considérera deux catégories d'ordinateur, une première qui, avant de « booter », demande un mot de passe, et une autre qui, avant de « booter », demande quel système d'exploitation on désire lancer. Esquissez le code des trois classes correspondantes.

Exercice 12.3

Tentez de prédire ce que le code Java suivant fera apparaître à l'écran. Réalisez le diagramme de classe UML correspondant.

```
class Electeur {
 private int age;
 private String adresse;
 protected Candidat[] lesCandidats;
 public Electeur(Candidat[] lesCandidats) {
  this.lesCandidats = lesCandidats;
 }
 public void jeVote() {}
}
class ElecteurIdiot extends Electeur {
 private int QI;
 public ElecteurIdiot(Candidat[] lesCandidats, int QI) {
  super(lesCandidats);
  this.QI = QI;
 }
 public void jeVote() {
  for (int i=0; i<lesCandidats.length; i++) {
   if ((lesCandidats[i].donneQI() > QI)
     || (lesCandidats[i].compareSlogan("vive la France, la semaine des 5 heures")))
   {
    lesCandidats[i].accroitVoix();
    break;
   }
  }
 }
}
```

```
class ElecteurIndecis extends Electeur {
 int age;
 public ElecteurIndecis(Candidat[] lesCandidats, int age) {
  super(lesCandidats);
  this.age = age;
 }
 public void jeVote() {
  for (int i=0; i<lesCandidats.length; i++) {
   if ((lesCandidats[i].getAge() < age)
     && ((lesCandidats[i].compareSlogan("vive la France, l'etat c'est moi"))
      ||
      (lesCandidats[i].compareSlogan("vive la France, etranger dehors"))
      ||
      (lesCandidats[i].compareSlogan("vive la France, regardez mon bilan"))))
   {
    lesCandidats[i].accroitVoix();
    break;
   }
  }
 }
}
class ElecteurMalin extends Electeur {
 public ElecteurMalin(Candidat[] lesCandidats) {
  super(lesCandidats);
 }
 public void jeVote() {
  for (int i=0; i<lesCandidats.length; i++) {
   if (lesCandidats[i].compareSlogan("vive la France, regardez mon bilan")) {
    lesCandidats[i].accroitVoix();
    break;
   }
  }
 }
}
class Candidat {
 private String nom;
 private int age;
 private int nbreCasseroles;
 private int QI;
 private int nombreDeVoix;

 public Candidat(String nom, int age, int nbreCasseroles, int QI) {
  this.nom = nom;
  this.age = age;
  this.nbreCasseroles = nbreCasseroles;
  this.QI = QI;
  nombreDeVoix = 0;
 }
 public int getAge() {
  return age;
 }
```

```
 public String monSlogan() {
  return "vive la France, ";
 }
 public void accroitVoix() {
  nombreDeVoix ++;
 }
 public int donneNombreCasseroles() {
  return nbreCasseroles;
 }
 public int donneQI() {
  return QI;
 }
 public boolean compareSlogan(String unSlogan) {
  /* la methode String.compareTo(String) renvoie 0 seulement si
    les deux strings sont egaux */
  if (unSlogan.compareTo(monSlogan())==0)
   return true;
  else
   return false;
 }
 public void donneNombreVoix() {
  System.out.println(nom + " a fait " + nombreDeVoix + " voix");
 }
}
class CandidatDangereux extends Candidat {
 public CandidatDangereux(String nom, int age, int nbreCasseroles, int QI) {
  super(nom,age,nbreCasseroles,QI);
 }
 public String monSlogan() {
  return super.monSlogan() + "etranger dehors";
 }
}
class CandidatEgoTrip extends Candidat {
 public CandidatEgoTrip(String nom, int age, int nbreCasseroles, int QI) {
  super(nom,age,nbreCasseroles,QI);
 }
 public String monSlogan() {
  return super.monSlogan() + "l'etat c'est moi";
 }
}
class CandidatBrillant extends Candidat {
 public CandidatBrillant(String nom, int age, int nbreCasseroles, int QI) {
  super(nom,age,nbreCasseroles,QI);
 }
 public String monSlogan() {
  return super.monSlogan() + "regardez mon bilan";
 }
}
class CandidatCasserole extends Candidat {
 public CandidatCasserole(String nom, int age, int nbreCasseroles, int QI) {
  super(nom,age,nbreCasseroles,QI);
 }
```

```
 public String monSlogan() {
  return super.monSlogan() + "regardez mon compte en banque";
 }
}
public class Exo3 {
 public static void main(String[] args) {
  Candidat[] lesCandidats = new Candidat[8];
  Electeur[] lesElecteurs = new Electeur[10];

  lesCandidats[0] = new CandidatDangereux("LePon",75,1000,50);
  lesCandidats[1] = new CandidatDangereux("Laguillerette",55,0,10);
  lesCandidats[2] = new CandidatDangereux("StChasse",50,100,10);
  lesCandidats[3] = new CandidatEgoTrip("LeChe",60,0,150);
  lesCandidats[4] = new CandidatEgoTrip("Madeleine",55,0,100);
  lesCandidats[5] = new CandidatCasserolle("SuperLier",70,1000,100);
  lesCandidats[6] = new CandidatBrillant("Jaudepis",65,0,1000);
  lesCandidats[7] = new CandidatBrillant("Tamere",55,0,800);

  lesElecteurs[0] = new ElecteurMalin(lesCandidats);
  lesElecteurs[1] = new ElecteurMalin(lesCandidats);
  lesElecteurs[2] = new ElecteurIndecis(lesCandidats, 20);
  lesElecteurs[3] = new ElecteurIndecis(lesCandidats,80);
  lesElecteurs[4] = new ElecteurIndecis(lesCandidats,60);
  lesElecteurs[5] = new ElecteurIndecis(lesCandidats,70);
  lesElecteurs[6] = new ElecteurIndecis(lesCandidats,40);
  lesElecteurs[7] = new ElecteurIdiot(lesCandidats,20);
  lesElecteurs[8] = new ElecteurIdiot(lesCandidats,10);
  lesElecteurs[9] = new ElecteurIdiot(lesCandidats,60);

  for (int i=0; i<lesElecteurs.length; i++)
   lesElecteurs[i].jeVote();
  for (int i=0; i<lesCandidats.length; i++)
   lesCandidats[i].donneNombreVoix();
 }
}
```

Exercice 12.4

Que donne l’exécution du programme Java suivant ?

```
// fichier A.java
public class A {
 public A() {}
}
// fichier B.java
public class B extends A {
 public B() {
  super();
 }
 public String toString() {
  return(" Hello " + super.toString());
 }
```

```
}
// fichier testAB.java
public class TestAB {
 public TestAB() {
  A a = new B();
  System.out.println(a);
 }
 public static void main(String[] args) {
  TestAB tAB = new TestAB();
 }
}
```

Exercice 12.5

Supprimez dans le code qui suit les lignes qui provoquent une erreur et indiquez si l’erreur se produit à la compilation ou à l’exécution. Quel est le résultat de l’exécution qui s’affiche à l’écran après suppression des instructions à problème ?

```
class A {
 public void a() {
    System.out.println("a de A") ;
}
public void b()  {
   System.out.println("b de A") ;
 }
}
class B extends A {
   public void b()  {
    System.out.println("b de B") ;
   }
   public void c()  {
      System.out.println("c de B") ;
   }

}
public class Correction2 {
   public static void main(String[] args)  {
         A a1=new A() ;
         A b1=new B() ;
         B a2=new A() ;
         B b2=new B() ;
         a1.a() ;
         b1.a() ;
         a2.a() ;
         b2.a() ;
         a1.b() ;
         b1.b() ;
         a2.b() ;
         b2.b() ;
         a1.c() ;
         b1.c() ;
```

```
          a2.c() ;
          b2.c() ;
          ((B)a1).c() ;
          ((B)b1).c() ;
          ((B)a2).c() ;
          ((B)b2).c() ;
               }
}
```

Exercice 12.6

Que donne l'exécution du programme C++ suivant ? Réalisez le diagramme de classe UML correspondant.

```
#include "stdafx.h"
#include "iostream.h"
class Animaux {
 private:
  int age;
  int id;
  void dormirEnFonctionDeMonAge() {
   if (age > 2)
    cout << "Je fais un petit ";
   else
    cout << "Je fais un gros ";
  }
  virtual void dormirAToutAge() {
   cout << "ronflement";
  }
 public:
  Animaux(int _age, int _id) : age(_age), id(_id) {}
  void dormir() {
   dormirEnFonctionDeMonAge();
   dormirAToutAge();
  }
};
class Employe {
 private:
  int age;
  int id;
  char *nom;
  void mangerDeTouteFacon() {
   cout <<"Je mange beaucoup de ";
  }
  virtual void mangerDifferemment() {
   cout <<"mes Animaux";
  }
 public:
  Employe(int _age, int _id, char* _nom): age(_age),id(_id) {
   nom = _nom;
  }
  void manger() {
   mangerDeTouteFacon();
   mangerDifferemment();
  }
```

```
};
class Elephant: public Animaux {
 public:
  Elephant(int _age, int _id):Animaux(_age,_id) {};
 private:
  void dormirAToutAge() {
   cout << "barrissement";
  }
};
class Lion: public Animaux {
 public:
  Lion(int _age, int _id):Animaux(_age,_id){}
 private:
  void dormirAToutAge() {
   cout << "rugissement";
  }
};
class Singe: public Animaux {
 public:
  Singe(int _age, int _id) : Animaux(_age,_id){}
};
class EmployeDuZoo: public Employe {
 public:
  EmployeDuZoo(int _age,int _id,char* _nom):Employe(_age,_id,_nom){}
  void mangerDeTouteFacon() {
   cout << "Je mange enormement de ";
  }
  void mangerDifferemment() {
   cout << "choucroute";
  }
};
class MandaiDuZoo: public Employe {
 public:
  MandaiDuZoo (int _age, int _id, char* _nom):Employe(_age,_id,_nom){}
  void mangerDeTouteFacon() {
   cout << "Je mange tres peu de ";
  }
  void mangerDifferemment() {
   cout <<"radis beurre";
  }
};
class ResponsableDuZoo:public Employe {
 private:
  Employe* mesEmployes[3];
  Animaux* mesAnimaux[3];
```

```
 public:
  ResponsableDuZoo (int _age, int _id, char* _nom,Employe* _mesEmployes[3],
    Animaux* _mesAnimaux[3]):Employe(_age,_id,_nom) {
     for (int i=0; i<3; i++) {
      mesAnimaux[i] = _mesAnimaux[i];
      mesEmployes[i] = _mesEmployes[i];
     }
    }
    void mangerDifferemment() {
     cout << "caviar";
    }
    void faireLaTourneeDuSoir() {
     for (int i=0; i<3; i++) {
      mesAnimaux[i]->dormir();
      cout << endl;
     }
     for (int j=0; j<3; j++) {
      mesEmployes[j]->manger();
      cout << endl;
     }
    }
};
int main(int argc, char* argv[]) {
 Employe *mesEmployes[3];
 mesEmployes[0]  = new EmployeDuZoo(30,2,"Dupont");
 mesEmployes[1]  = new EmployeDuZoo(25,3,"Durant");
 mesEmployes[2]  = new MandaiDuZoo(23,4,"Michel");
 Animaux *mesAnimaux[3];
 mesAnimaux[0]   = new Lion(1,0);
 mesAnimaux[1]   = new Elephant(3,1);
 mesAnimaux[2]   = new Singe(2,2);
 ResponsableDuZoo JeanMarie(60,1,"JeanMarie",
   (Employe*[3])mesEmployes,(Animaux*[3])mesAnimaux);
 JeanMarie.faireLaTourneeDuSoir();
 return 0;
}
```

Exercice 12.7

Que donne l’exécution du programme C# suivant ?

```
using System;

class InstrumentDeMusique {
 private String nomInstrument;
 private double poidsInstrument;
 private Musicien joueurInstrument;
 private static int nombreInstrumentDansOrchestre;
```

```
 public InstrumentDeMusique(String nomInstrument, double poidsInstrument,
   Musicien joueurInstrument) {
  this.nomInstrument   = nomInstrument;
  this.poidsInstrument  = poidsInstrument;
  this.joueurInstrument  = joueurInstrument;
  nombreInstrumentDansOrchestre ++;
 }
 public double donneMonPoids() {
  return poidsInstrument;
 }
 public Musicien donneMonJoueur() {
  return joueurInstrument;
 }
 public Boolean seraiJeBienJoue() {
  if (joueurInstrument.donneExperience() > 10)
   return true;
  else
   return false;
 }
 public virtual void testDesaccordage() {
  Console.WriteLine("on teste");
 }
}
class Violon : InstrumentDeMusique {
 private String marqueDesCordes;
 private int frequenceDaccordage;
 private int temperatureLimite;
 private int temperatureAmbiante;
 private static int nombreViolonDansOrchestre;

 public Violon(String nomInstrument, double poidsInstrument,
   Musicien joueurInstrument, String marqueDesCordes,
   int temperatureAmbiante)
  :base(nomInstrument, poidsInstrument, joueurInstrument)
 {
  this.marqueDesCordes    = marqueDesCordes;
  frequenceDaccordage     = 12;
  temperatureLimite      = 40;
  this.temperatureAmbiante  = temperatureAmbiante;
 }
 public override void testDesaccordage() {
  if (temperatureAmbiante > temperatureLimite)
   Console.WriteLine("Attention violon desaccorde");
  else
   Console.WriteLine("Tout va bien");
 }
}
class Piano : InstrumentDeMusique {
 private int frequenceDaccordage;
 private String nomDeLaccordeur;
```

```
  private static int nombrePianoDansOrchestre;

  public Piano(String nomInstrument, double poidsInstrument,
    Musicien joueurInstrument, String nomDeLaccordeur)
   :base(nomInstrument, poidsInstrument, joueurInstrument)
  {
   frequenceDaccordage  = 24;
   this.nomDeLaccordeur = nomDeLaccordeur;
  }
  public String donneNomAccordeur() {
   return nomDeLaccordeur;
  }
  public override void testDesaccordage() {
   if (nomDeLaccordeur == "")
    Console.WriteLine("Attention pas d'accordeur de piano");
   else
    Console.WriteLine("Tout va bien");
  }
 }
 class Musicien {
  private String nom;
  private int experience;
  private int age;

  public Musicien(String nom, int experience, int age) {
   this.nom        = nom;
   this.experience = experience;
   this.age        = age;
  }
  public String donneNome() {
   return nom;
  }
  public int donneAge() {
   return age;
  }
  public int donneExperience() {
   return experience;
  }
 }
 public class Exo7 {
  public static void Main() {
   InstrumentDeMusique[] lesInstruments = new InstrumentDeMusique[4];
   Musicien[] lesMusiciens = new Musicien[4];

   lesMusiciens[0]    = new Musicien("Pat",5,25);
   lesMusiciens[1]    = new Musicien("Herbie",15,22);
   lesMusiciens[2]    = new Musicien("Brad",15,34);
   lesMusiciens[3]    = new Musicien("Joe",5,18);

   lesInstruments[0]  = new  Violon("stradivarius",2,lesMusiciens[0],"cordeMeilleure",42);
   lesInstruments[1]  = new Piano("playel11",150,lesMusiciens[1],"");
```

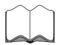
```
  lesInstruments[2]  = new Piano("Playel12",135,lesMusiciens[2],"Albert");
  lesInstruments[3]  = new InstrumentDeMusique("Instrument",200,lesMusiciens[1]);

  for (int i=0; i<4; i++)
   lesInstruments[i].testDesaccordage();
 }
}
```

13

Abstraite, cette classe est sans objet

Ce chapitre introduit la notion de classe abstraite et son exploitation lors du polymorphisme.

Sommaire : Classe abstraite — Méthode abstraite, virtuelle pure — Polymorphisme, encore — Les interfaces graphiques



Doctus — Les caractéristiques de nos objets sont un moyen de les identifier. L'ensemble de leurs méthodes permet de savoir ce qu'ils représentent et ce qu'ils font. Elles permettent même de définir nos objets.

Candidus — Nous avons déjà parlé de ça, où veux-tu en venir ?

Doc. — Rappelle-toi que la définition d'une méthode est constituée de sa signature : type retourné, nom de méthode et liste de ses arguments. Son implémentation est l'affaire du mécanisme d'héritage. Nous pouvons donc nous contenter, dans un premier temps, de déclarer l'existence de certaines méthodes tout en remettant à plus tard leur réalisation concrète.

Cand. — Et qu'est-ce qu'on y gagne ?

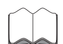
Doc. — Cela revient à dire que les objets savent faire certaines choses mais sans dire tout de suite comment.

Cand. — Je pourrai donc créer une classe d'objets comme je le fais d'habitude mais, je pourrai, pour certaines méthodes, dire que l'implémentation doit être recherchée dans le type concret de l'objet ?

Doc. — Il vaut mieux préciser que tu diras comment ailleurs plutôt que tout de suite. C'est en fait dans des sous-classes, bien concrètes celles-la, que tu devras réaliser les méthodes concernées.

Cand. — Nos classes concrètes représentent les différentes formes que peuvent prendre les objets qu'on manipule par leur poignée abstraite, c'est ça ?

Doc. — Ta poignée s'appelle une *classe abstraite*. Tu pourras t'en servir pour manipuler ces objets mais elle ne sera, du moins en partie, qu'une sorte de squelette que tu utiliseras pour regrouper tout ce qui est concret et commun à un groupe d'objets, tout en mentionnant des méthodes abstraites que tu te proposes de réaliser dans des sous-classes qui vont en hériter.



De Canaletto à Turner

Le peintre vénitien Canaletto a peint de multiples vues de Venise au XVIIIe siècle. Elles sont extraordinaires par la précision et la profusion de détails qui nous sont rapportés. Canaletto réalisait ses œuvres afin de satisfaire des commandes de notables anglais, exigeant une vue précise de Venise, non avare de détails, sous la forme d'un reportage fidèle à la réalité. Il y a bien évidemment une « aspiration photographique » dans ce travail. Elles sont précises au point que, grâce à elles, on a pu déduire exactement de quelle hauteur, depuis l'époque du peintre, les eaux avaient monté dans Venise.

Turner a lui aussi peint Venise quelque 100 ans plus tard. Venise y est moins nette, bien qu'on en devine les caractéristiques essentielles. Nous sommes aux sources de l'abstraction picturale, où la peinture exprime davantage la vision intérieure de l'artiste que la réalité. Il cherche à communiquer sa Venise à lui, et, ce faisant, à suggérer les émotions qu'elle provoque en lui. Néanmoins, cette abstraction conserve de nombreux traits de Venise, faisant l'économie de leur implémentation détaillée. Venise est entre les lignes. C'est la signature de Venise, bien plus que sa photo. Cela présente l'avantage de bien mieux vieillir et de ressembler à Venise aujourd'hui, bien plus que l'œuvre de Canaletto, qui n'est plus à jour. Les abstractions tiennent mieux la distance. Il en va un peu ainsi des classes abstraites par rapport aux classes concrètes. Dans une des vues de Venise de Turner, on devine un petit personnage peignant dans un coin. Vous aurez deviné de qui il s'agit.

Des classes sans objet

En se replongeant dans les deux petits programmes illustrant les mécanismes d'héritage : l'écosystème et le match de football, force serait de faire le constat suivant : dès qu'une superclasse apparaît dans le code, elle n'a plus l'occasion de donner naissance à des objets. Dans le code de l'écosystème, ne figure aucun objet de type faune ou ressource, et dans le match de football ne joue aucun joueur. Au moment de la création de l'objet à proprement parler, lorsque les joueurs montent sur le terrain, lors de l'utilisation du « new », la classe qui suit ce « new » et qui type dynamiquement l'objet n'a plus lieu d'être une superclasse.

Vous aurez tôt fait de nous rétorquer qu'en étant instance de la sous-classe, tous les objets le sont automatiquement de la superclasse. C'est exact, d'un point de vue déclaration, ou typage statique, et c'est vrai pour le compilateur (ce qui n'est déjà pas rien), mais cela ne reste que partiellement vrai lors de l'exécution. Les objets sont d'abord d'un type dynamique avant d'être également du type statique, comme nombre d'immigrés vous diront qu'ils sont d'abord français (ou devenus tels) avant d'être italien, algérien, polonais ou argentin. Tout objet peut être de plusieurs types statiques, hérités de leurs parents et grands-parents, mais ne sera que d'un, et un seul, type dynamique, sa véritable et ultime nature.

Rien n'interdit, pour l'instant, de créer dynamiquement des objets de type superclasse. Mais on conçoit aisément que, dès que l'univers conceptuel qui nous intéresse est couvert de sous-classes, c'est-à-dire, et pour reprendre la théorie des ensembles, lorsque chaque élément de l'ensemble est repris dans un sous-ensemble, il ne soit plus justifié de créer encore des objets de la superclasse. Votre voiture est une Renault avant d'être une voiture, votre chien est un cocker avant d'être un chien, le joueur de football est un attaquant ou un défenseur avant d'être un joueur. Cela pourrait néanmoins être le cas, si on vous demande de rajouter un animal dans votre logiciel sans préciser son espèce, ou si on vous offre une voiture sans préciser sa marque, ou si l'entraîneur décide d'envoyer un joueur sur le terrain sans lui indiquer quel poste il occupe, mais c'est plutôt rare. On sait pertinamment de quelle nature intime sont les objets auxquels on a affaire. Dans la pratique courante de l'OO, les superclasses, bien qu'indispensables à la factorisation des caractéristiques communes aux sous-classes, ne donnent que très rarement naissance à des objets. La figure 13-1 dépeint la situation idéale dans laquelle les objets sont tous issus d'une sous-classe ; aucun objet n'est laissé au niveau de la super classe.

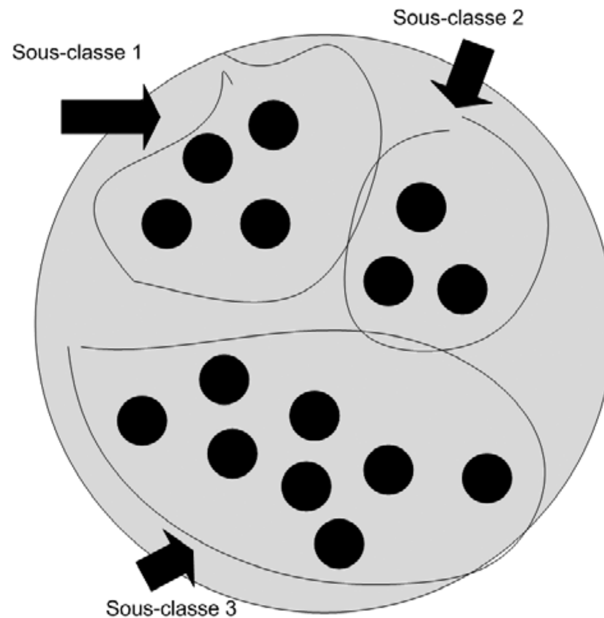


Figure 13-1
Une situation d'héritage idéale, avec tous les objets issus des seules sous-classes

Du principe de l'abstraction à l'abstraction syntaxique

Ayez à l'esprit que ce ne serait pas une bourde syntaxique de créer des objets instances d'une superclasse, sauf dans un cas précis, que nous allons maintenant détailler, et qui se produit quand vous déclarez explicitement la superclasse comme étant « abstraite ». Nous nous baserons pour comprendre la nature et le rôle des classes abstraites sur le modèle de l'écosystème. Dans le code, la classe `Jungle` envoie de manière répétée le même message `evolue()` aux objets issus des deux sous-classes de `Ressource` : `Eau` et `Plante`. L'exécution de ce message n'a à ce point rien de commun entre l'eau (elle s'assèche) et la plante (elle pousse) qu'aucun corps d'instruction n'est repris dans la classe `Ressource`. L'eau et la plante, bien qu'évoluant toutes deux, et capable de recevoir ce même message, d'où qu'il provienne, ne partagent rien dans l'exécution de celui-ci.

Dans le code Java qui suit, tant la classe `Eau` que la classe `Plante` intègrent la méthode `évolue()` :

```
public class Plante {          public class Eau {
  ……   ……..
  ……   ……..
                                ……   …….
 public void evolue() {         public void evolue() {
                                  …………………….
     ………                            ………
    ………..                          ………..
  }                                   }
}                              }
```

Vous pourriez décemment vous demander à quoi cela sert de nommer ces méthodes de la même manière, si elles décrivent des réalités si distinctes. Rappelez-vous ce que nous vous disions sur la pauvreté de notre langage, quand il s'agit de décrire des modalités actives par rapport aux modalités structurelles.

Voilà une première raison. Il en est une seconde qui tient plus à la pratique logicielle. Il est intéressant de pouvoir écrire le code de la jungle, la « tierce » classe, « cliente » de l’eau et de la plante, comme envoyant indifféremment un même message aux points d’eau et aux plantes. Une économie d’écriture sera véritablement réalisée s’il est possible, à l’instar des joueurs de football recevant le message `avance()` de l’entraîneur, de permettre à la `Jungle` d’envoyer le même message `évolue()`, en boucle, à toutes les ressources auxquelles elle est associée (sans se préoccuper du nombre et de la nature de celles-ci), comme ci-après :

```
for (int i=0; i<lesRessources.length; i++)
  lesRessources[i].evolue();
```

On pourrait imaginer créer un ensemble de 100 points d’eau, par la simple instruction :

```
for (int i=0 ; i<100 ; i++)
  lesRessources[i] = new Eau() ;
```

Et 200 plantes, au moyen de :

```
for (int i=0 ; i<200 ; i++)
  lesRessources[100+i] = new Plante() ;
```

Et d’envoyer ensuite le message `évolue()` sur ces 300 ressources. C’est en effet ce que l’on cherche à faire, en tous les cas, au moment de l’exécution du code. On désirerait ajouter un nouveau type de ressource, par exemple, des cadavres en décomposition d’autres animaux, que le code de la jungle ne se modifierait en rien.

Malheureusement pour nous (mais heureusement dans pratiquement tous les cas de figure), l’exécution est toujours précédée par une étape de compilation (comme nous le savons, Python et PHP 5 se distinguent ici) qui, parmi d’autres choses, fait office de correcteur syntaxique plutôt sévère. Or, comme dans tous les langages de programmation, un tableau se doit d’être typé. Si nous voulons donc installer toutes les ressources dans un tableau, il faudra typer ce dernier au moyen d’une instruction telle que :

```
Ressource [] lesRessources = new Ressource[300].
```

Pouvions-nous typer ce tableau comme `Plante` ? Non, car il y a des points d’eau dans l’affaire. Et comme `Eau` ? Non, car il y a des plantes dans l’affaire. La seule solution est de le typer comme `Ressource`, puisqu’en effet, tant les plantes que les eaux en sont. Et nous nous retrouvons, comme dans le chapitre précédent, en présence d’objet dont le type statique, `Ressource`, diffère du type dynamique : `Eau` ou `Plante`. Nous nageons à nouveau, avec bonheur, en plein polymorphisme.

La classe `Jungle` pourrait également recevoir dans une de ses méthodes un argument de type `Ressource`, sur lequel elle enverrait un message commun à la plante ou l’eau, et qui serait par la suite exécuté différemment. Une nouveauté, essentielle ici, est que ni la classe `Eau` ni la classe `Plante` ne redéfinisse une méthode `evolue()`, qui aurait une part d’instructions déjà prévue dans la superclasse. Pourtant, si nous typons le tableau des ressources comme `Ressource`, et que nous envoyons le message « évolue » sur chacun de ces objets, le compilateur, pour qui seul le type statique a voix au chapitre, ne pourra accepter qu’aucune méthode `evolue()` ne soit en effet présente dans la classe `Ressource`.

Dilemme, dont la seule issue possible est d’installer une méthode `evolue()` dans la classe `Ressource`, tout en déclarant cette méthode « abstraite », c’est-à-dire sans corps d’instruction. Une méthode abstraite est une méthode qui se limite à sa seule signature, une méthode qui ne fait rien, à part se présenter.

En Java, en C# et en PHP 5, nous la déclarons dans la classe `Ressource` de la manière suivante :

```
abstract public void evolue();
```

En C++, elle serait dite méthode « virtuelle pure », et se déclare ainsi :

```
public :
  void virtual evolue() = 0;
```

Elle est virtuelle par la présence de `virtual`. En ajoutant `= 0`, on la rend abstraite. Nous verrons par la suite une manière de réaliser l'abstraction dans Python.

Classe abstraite

Toute classe contenant au moins une méthode abstraite devient d'office abstraite. D'ailleurs, tant Java que C# et PHP 5 forcent le trait, en vous obligeant à rajouter le mot-clé `abstract` dans la déclaration de la classe, comme suit :

```
public abstract class Ressource extends ObjetJungle {
      …….
}
```

C++ reste plus sobre et sait que l'abstraction d'au moins une méthode entraîne l'abstraction de toute la classe. Il n'y a d'autre moyen de rendre une classe abstraite qu'en y installant une méthode abstraite ou virtuelle pure. En fait, Java, C# et PHP 5 n'accepteraient pas qu'une méthode abstraite ne fût définie dans une classe, elle-même déclarée comme abstraite, mais le contraire ne s'applique pas. Les trois langages acceptent d'une classe qu'elle soit abstraite, alors qu'aucune méthode abstraite ne s'y trouve. Ils bloquent ainsi la possibilité pour certaines classes de donner naissance à des objets, indifféremment du fait qu'elles intègrent une méthode abstraite. Dans la pratique, très logiquement, une classe ne sera généralement abstraite que si une méthode abstraite s'y trouve.

« new » et « abstract » incompatibles

Au début de ce chapitre, nous vous expliquions que, souvent, les superclasses ne donnent pas naissance à des objets. Dorénavant, elles le pourront d'autant moins qu'elles seront déclarées abstraites. `new` et `abstract` sont deux mots-clés totalement incompatibles, en ce sens qu'aucune allocation de mémoire ne peut être effectuée pour des instances de classe abstraite. Si nous revenons à la définition première des classes abstraites, c'est-à-dire qu'elles contiennent au moins une méthode abstraite, cette interdiction doit vous paraître logique.

Supposons une classe contenant une méthode abstraite et pouvant donner naissance à des objets. Tout objet se doit être capable d'exécuter tous les messages reçus. Qu'en serait-il du message issu de la méthode abstraite ? Le compilateur ne tiquerait pas, car la syntaxe du message est parfaitement correcte. Mais que faire à l'exécution, face à un corps d'instruction absent ? On enverrait un message qui dit de ne rien faire ? Cette possibilité a d'office été bannie par les langages OO, car un message se doit de faire quelque chose.

Notez pour l'anecdote qu'un corps d'instruction vide est considéré comme distinct de pas de corps d'instruction du tout : `public void evolue() {}` est différent de `abstract public void evolue()`. Tous les langages OO interdisent l'envoi de messages à partir de méthodes sans corps d'instruction, mais cette interdiction est levée pour des méthodes dont le corps d'instruction, bien qu'existant, est vide. Seules les premières méthodes sont abstraites, les autres sont stupides mais concrètes !

Abstraite de père en fils

Au contraire des superclasses concrètes, les superclasses abstraites obligent à redéfinir (ne serait-il sans doute pas plus approprié de simplement dire « définir » ?) les méthodes abstraites dans leurs sous-classes. Tant que la méthode abstraite n'est pas redéfinie dans les sous-classes, chacune de ces sous-classes se doit de rester abstraite,

et aucune ne donnera naissance au moindre objet. Le compilateur se chargera de vérifier que vous maintenez l’abstraction, de sous-classes en sous-classes, jusqu’à ce que toutes les méthodes abstraites soient redéfinies.

Ci-après, vous voyez le code Java de la superclasse abstraite `Ressource` et de la sous-classe concrète `Eau`. La méthode `dessineToi()`, qui représente graphiquement la ressource, est abstraite dans la classe `Ressource`, car il est nécessaire de savoir de quelle ressource il s’agit avant de la dessiner. Tous les objets se dessinent, mais tous le feront à leur manière. La méthode `evolue()`, pour des raisons déjà évoquées, est également abstraite. Les plantes évoluent en grandissant, les points d’eau en diminuant de taille.

```
public abstract class Ressource extends ObjetJungle { /* classe abstraite */
  private int temps;
  private int quantite;

  Ressource () {
    super();
    temps = 0;
    quantite = 100;
  }
  abstract public void dessineToi(Graphics g);  /* méthode abstraite */
  abstract public void evolue();                /* méthode abstraite */
  public void incrementeTemps() {
    temps++;
  }
  public int getTemps() {
    return temps;
  }
  public void diminueQuantite() {
    decroitTaille(2);
  }
}
public class Eau extends Ressource {
  Eau() {
    super();
  }
  public void dessineToi(Graphics g) {  /* définition de cette méthode abstraite */
    g.setColor(Color.blue);
    g.fillOval(getMaZone().x, getMaZone().y, getMaZone().width, getMaZone().height);
  }
  public void evolue() {                /* définition de cette méthode abstraite */
    incrementeTemps();
    if ((getTemps()%10) == 0)
      decroitTaille(2);
  }
}
```

Un petit exemple dans les cinq langages de programmation

Ci-après, vous trouverez en Java, C#, PHP 5 et C++ un même exemple d’une superclasse abstraite, dû à la présence en son sein d’une méthode abstraite, ainsi que deux sous-classes concrétisant cette même méthode de deux manières différentes.

En Java

```
abstract class O1 {
  abstract public void jexisteSansRienFaire();
}
class FilsO1 extends O1 {
  public void jexisteSansRienFaire() {
    System.out.println("ce n'est pas vrai, je fais quelque chose");
  }
}
class AutreFilsO1 extends O1 {
  public void jexisteSansRienFaire() {
    System.out.println("c'est de nouveau faux, moi aussi je fais quelque chose");
  }
}
public class ExempleAbstract {
  public static void main(String[] args) {
    /* O1 unO1 = new O1(); impossible */
    O1 unFilsO1         = new FilsO1();
    O1 unAutreFilsO1    = new AutreFilsO1();
    unFilsO1.jexisteSansRienFaire();
    unAutreFilsO1.jexisteSansRienFaire();
  }
}
```

Résultat

```
ce n’est pas vrai, je fais quelque chose
c’est de nouveau faux, moi aussi je fais quelque chose
```

Remarquez que nous avons délibérément typé statiquement et dynamiquement nos objets de manière différente, le type statique ne pouvant être qu’une superclasse du type dynamique. Alors qu’il n’est pas possible de typer dynamiquement un objet par une classe abstraite, comme le montre le code (impossible de créer un objet comme étant typé « définitivement » par une classe abstraite), il n’y a aucun problème pour le typer statiquement avec une classe abstraite. C’est de fait une pratique très courante et inhérente au polymorphisme.

En C#

```
using System;
abstract class O1 {
  abstract public void jexisteSansRienFaire();
}
class FilsO1 : O1 {
  public override void jexisteSansRienFaire() {
    Console.WriteLine("ce n'est pas vrai, je fais quelque chose");
  }
}
```

```
class AutreFils01 : O1{
  public override void jexisteSansRienFaire() {
    Console.WriteLine("c'est de nouveau faux, moi aussi je fais quelque chose");
  }
}
public class ExempleAbstract {
  public static void Main() {
    /* O1 unO1 = new O1(); impossible */
    O1 unFils01 = new Fils01();
    O1 unAutreFils01 = new AutreFils01();
    unFils01.jexisteSansRienFaire();
    unAutreFils01.jexisteSansRienFaire();
  }
}
```

Résultat

```
ce n’est pas vrai, je fais quelque chose
c’est de nouveau faux, moi aussi je fais quelque chose
```

Rien d’essentiellement différent par rapport au code Java, si ce n’est l’addition du mot-clé `override`, lors de la concrétisation des méthodes abstraites. Le mot-clé `virtual`, lors de la déclaration des méthodes abstraites, n’est plus nécessaire, comme il l’est lors de la déclaration des méthodes, non plus abstraites, mais concrètes et à redéfinir.

En PHP 5

```
<html>
<head>
<title> Héritage et abstraction </title>
</head>
<body>
<h1> Héritage et abstraction </h1>
<br>

<?php
     abstract class O1 {
          abstract public function jexisteSansRienFaire();
     }

     class Fils01 extends O1 {
          public function jexisteSansRienFaire() {
               print ("ce n'est pas vrai, je fais quelque chose <br> \n");
          }
     }

     class AutreFils01 extends O1 {
          public function jexisteSansRienFaire() {
               print ("c'est de nouveau faux, moi aussi je fais quelque chose <br> \n");
          }
     }

     $unFils01 = new Fils01();
     $unAutreFils01 = new AutreFils01();
     $unFils01->jexisteSansRienFaire();
```

```
    $unAutreFils01->jexisteSansRienFaire();
?>
</body>
</html>
```

Point de typage statique différent d'un typage dynamique, car point de compilation. Mais à cette différence essentielle près, l'abstraction se réalise comme dans les deux langages précédents (et elle est plutôt très inspirée de Java, comme l'est toute la partie « héritage » de PHP 5).

En C++

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
  public:
    virtual void jexisteSansRienFaire() = 0;
};
class Fils01 : public O1 {
  public:
    void jexisteSansRienFaire() {
      cout <<"ce n'est pas vrai, je fais quelque chose" << endl;
    }
};
class AutreFils01 : public O1 {
  public:
    void jexisteSansRienFaire() {
      cout <<"c'est de nouveau faux, moi aussi je fais quelque chose" << endl;
    }
};
int main(int argc, char* argv[]) {
  Fils01 unFils01;
  AutreFils01 unAutreFils01;
  /* O1 unO1; impossible */
  unFils01.jexisteSansRienFaire();
  unAutreFils01.jexisteSansRienFaire();

  O1* unFils01Pointeur = new Fils01();
  O1* unAutreFils01Pointeur = new AutreFils01();

  unFils01Pointeur->jexisteSansRienFaire();
  unAutreFils01Pointeur->jexisteSansRienFaire();

  return 0;
}
```

Résultat

```
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
```

En C++, le mot-clé `abstract` disparaît. La déclaration de la méthode comme « virtuelle pure » suffit à rendre la classe abstraite. Nous présentons deux versions du programme, selon que les objets sont créés dans la mémoire pile ou la mémoire tas. Nous voyons que, dans le premier cas, il n'est pas possible de typer statiquement des objets par une classe abstraite car, comme dans ce cas, la seule déclaration suffit à la création de l'objet, les deux types se doivent d'être égaux. L'utilisation du polymorphisme à partir de classe abstaite n'est donc possible qu'avec des pointeurs et sur des objets installés dans la mémoire tas.

Classe abstraite

Une classe abstraite en Java, C##, PHP 5 et C++ (dans ce cas, en présence d'une méthode virtuelle pure) ne peut donner naissance à des objets. Elle a comme unique rôle de factoriser des méthodes et des attributs communs aux sous-classes. Si une méthode est abstraite dans cette classe, il sera indispensable de redéfinir cette méthode dans les sous-classes, sauf à maintenir l'abstraction pour les sous-classes et à opérer la concrétisation quelques niveaux en dessous.

L'abstraction en Python

Bien qu'il ne soit pas possible de déclarer une classe abstraitre en Python (à cause de la simplification de la syntaxe et de l'affaiblissement du typage), une petite pirouette, illustrée dans le code ci-dessous, permet d'empêcher la classe `O1` de donner naissance à des objets.

```
class O1:
       def __init__(self):
       assert self.__class__ is not O1
       def jexisteSansRienFaire(self):
               pass

class FilsO1(O1):
       def jexisteSansRienFaire(self):
               print "ce n'est pas vrai, je fais quelque chose"

class AutreFilsO1(O1):
       def jexisteSansRienFaire(self):
               print "c'est de nouveau faux, moi aussi je fais quelque chose"

# unO1=O1() devenu impossible
unFilsO1=FilsO1()
unAutreFilsO1=AutreFilsO1()
unFilsO1.jexisteSansRienFaire()
unAutreFilsO1.jexisteSansRienFaire()
```

L'intruction « assert » évalue la condition qui suit (ici, dans le constructeur, vérifie si l'objet en création n'est pas de la classe `O1`). Si cette condition est vérifiée, cette instruction ne fait rien. Dans le cas contraire, une exception est produite et levée (elle peut alors être `try-catch` comme expliqué dans le chapitre 7). La classe `O1` pourra être héritée et aura dès lors comme seul rôle de factoriser des méthodes à redéfinir dans les classes filles. Comme Python n'a que faire du typage statique, les classes abstraites ne joueront pas un rôle exactement semblable à celui joué dans les situations polymorphiques possibles et fréquentes dans les trois autres langages.

Un petit supplément de polymorphisme

Les enfants de la balle

Monsieur Loyal, dans son costume rouge mité, centré au milieu du disque lumineux, d'un revers de la main interrompt les cuivres et les cymbales un peu rouillés, et dans un éclat de voix strident hurle : « Que tous les artistes fassent leur numéro. » Et, bientôt, l'un après l'autre, tous les artistes viendront s'exécuter sur la piste. Mais ce qu'ils ne savent pas tous ces artistes, ces jongleurs, ces clowns, ces trapézistes, ces funambules et ces dompteurs, tous ces enfants ou, devrais-je dire, toutes ces sous-classes de la balle, c'est qu'ils feront leur numéro, ils le feront mais ne le feront pas à la manière Bouglione ou à la manière Pinter, ils le feront à la manière polymorphique.

Tous ont reçu cinq sur cinq le message de Monsieur Loyal, tous exécuteront leur numéro, tous l'exécuteront avec méthode, mais chacun à sa façon. Monsieur Loyal envoie un même message à un tableau d'artistes de cirque, artiste abstrait (la méthode `faireMonNuméro` étant abstraite dans la classe `Artiste`), et chacun se lancera dans le numéro qu'il a concrétisé et si longtemps répété dans sa sous-classe d'artiste, devenue pour le coup concrète, elle aussi : `Jongleur`, `Trapéziste`, `Dompteur`…

Il est important pour M. Loyal de savoir que la classe `Artiste` existe pour pouvoir interagir avec tous ces artistes d'une seule et même manière, quitte à ce que ce qui leur soit demandé se prête à une réalisation différente. S'il convoque un de ces artistes dans son bureau pour le payer, il lui enverra un message, ayant comme but l'établissement des prestations. Il est, là encore, bien possible que ces prestations doivent être établies différemment selon l'artiste en question. Un clown pourrait coûter moins cher qu'un dompteur ou un trapéziste, d'où sa tristesse.

Cliquez frénétiquement

Empoignez votre souris d'ordinateur, et cliquez frénétiquement, un clic, deux clics, cliquez où vous pouvez, cliquez où vous voulez, mais cliquez. Ce qui se produit sur l'écran, en réponse à ces clics, dépend de l'endroit où vous cliquez, de l'objet graphique sur lequel vous cliquez. Un menu apparaît, une fenêtre se ferme, une autre s'ouvre, un onglet passe au premier plan, une nouvelle police de caractère est mémorisée, le curseur se transforme en croix, etc. Il s'en passe des choses en réaction à ce clic, et pourtant le clic est toujours le même.

Parfois, vous pouvez juste le doubler rapidement, parfois votre souris possède un, deux ou trois boutons, et le clic se fait sur l'un ou l'autre. Cela laisse malgré tout très peu de modalités d'action, en comparaison au nombre d'objets graphiques qui réagiront différemment en réaction à ces quelques modalités d'action. Une poignée de modalités d'action, effectives sur une large panoplie d'objets, réagissant tous différemment en regard de ces actions, les interfaces graphiques des systèmes exploitations, Windows, Mac OS ou Linux, sont de merveilleux exemples de polymorphisme. Une souris, un clic, et une véritable taxonomie d'objets graphiques, capables de réagir à ce clic, voilà comment on peut résumer très schématiquement l'interaction de l'utilisateur avec ces systèmes d'exploitation.

Pourquoi ces objets graphiques sont-ils organisés de façon hiérarchique ? Car une fenêtre est un de ces objets qui peut se déplacer, s'agrandir, et possède un système de glissement qui permet de dévoiler, en partie seulement, la fenêtre. Une icône est un objet graphique qui peut juste se déplacer, un menu est un objet graphique qui ne peut pas se déplacer mais peut s'ouvrir, etc. Il est clair que certaines modalités d'action sont partagées par certains de ces objets et, ainsi, sont-elles factorisables dans des superclasses abstraites. D'autres seront spécifiées de manière polymorphique, au bas de l'arbre taxonomique, comme les effets de la souris. La figure 13-2 montre les différents objets graphiques Java, et leur structure d'héritage.

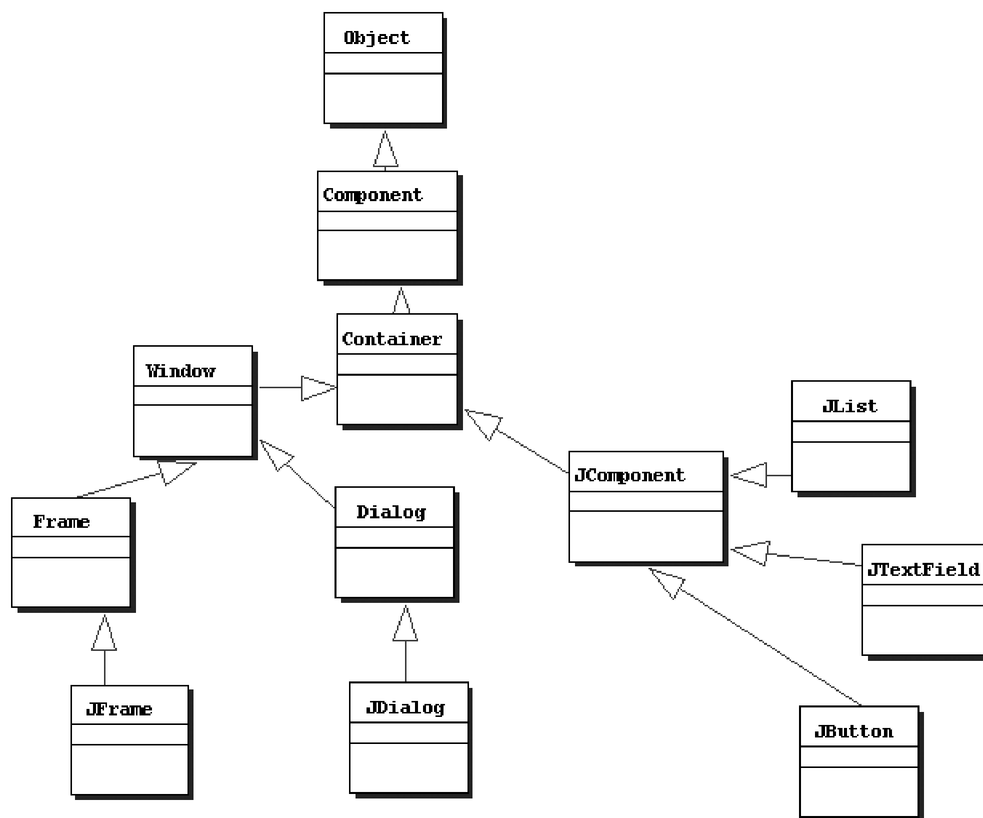


Figure 13-2
La hiérarchie des classes graphiques en Java.

On constate que Java, comme recommandé par la charte du bon programmeur objet, n'est pas économe des niveaux d'héritage. Ici on en décompte jusque six.

Ce détour par les interfaces graphiques des systèmes d'exploitation est loin d'être innocent, car l'histoire de l'orienté objet est concomitante, en partie, à l'histoire des interfaces graphiques. Lorsque Steve Jobs, célèbre gourou de l'informatique et créateur des Macintosh, se rend au Xerox PARC en 1979, Alan Kay, leader d'un groupe de recherche, lui présente les trois technologies innovantes sur lesquelles il planche. Tout d'abord, la mise en réseau des ordinateurs selon un protocole encore balbutiant : Internet. Steve Jobs n'y voit rien de très prometteur. Ensuite, une nouvelle manière de programmer, implémentée, en grande partie, dans un nouveau langage de programmation, inspiré de Simula, mais largement amélioré : Smalltalk.

À nouveau, Steve Jobs ne voit pas là de quoi fouetter un programmeur. Alan Kay décrit pourtant cette manière de programmer, OO comme il se doit, comme l'approche la plus élégante et la plus directe pour réaliser ce qui est son troisième domaine de recherche : la conception de nouvelles modalités d'interaction avec l'ordinateur : fenêtres, souris, menus… On connaît aujourd'hui cette musique par cœur, mais, en 1979, toute interaction se faisait *via* des lignes de commande tapées au clavier. En découvrant cette dernière recherche, Steve Jobs est subjugué, il n'en croit pas ses oreilles et ses yeux, et il comprend que c'est la manière la plus innovante et à la fois la plus naturelle de penser l'utilisation de l'ordinateur. Il part avec, sous le bras, son projet d'un nouveau système d'exploitation pour les Macs. On connaît la suite de l'histoire…

Un certain Bill Gates passa aussi par là, jeta un œil par la fenêtre, et Windows, comme par hasard, vi le jour. Chaque fois que vous cliquez à l'écran, ouvrez une fenêtre ou déroulez un menu, c'est en partie à Alan Kay que vous le devez.

Alan Kay

Il obtient son doctorat de l'université d'Utah en 1969 ; le sujet en est le développement d'interfaces graphiques 3-D. L'approche OO lui semble la plus naturelle pour la réalisation de ces interfaces graphiques. De ses 10 années passées au légendaire Xerox PARC (et il n'est pas pour rien dans la naissance de cette légende), il aura apporté une contribution essentielle à la mise au point du langage Smalltalk qui, bien que venant après Simula, est sans conteste le premier langage OO populaire et diffusé. Il participe aussi activement à la mise au point des protocoles réseaux Ethernet et Internet.

Son dernier sujet de préoccupation, et qui l'occupe encore activement aujourd'hui, est de repenser l'interaction homme-ordinateur, afin de ne pas laisser l'ordinateur se cantonner à une machine à écrire sophistiquée, mais de le transformer en un réel support à la créativité et l'imagination. C'est en observant, auprès de Seymour Papert, les enfants utiliser l'ordinateur, qu'il se rend compte qu'il y a dans cette « machine » un potentiel largement sous-exploité pour l'enrichissement intellectuel des enfants et des adultes en général. Depuis toutes ces années d'observation, l'éducation des enfants (dès l'âge de 6 ans), par le biais d'une utilisation mieux pensée des technologies de l'information, est devenue pour lui une croisade.

On sait tout ce que Steve Jobs lui doit et, de fait, il s'associe au succès d'Apple pendant les 10 années qui suivent. Il apporte sa touche essentielle dans la mise au point du Netwon d'Apple, mais le succès n'est pas au rendez-vous, car son rêve d'un système permettant une vraie interaction intuitive et créative, un compagnon aussi indispensable que discret, ne parvient à aboutir, suite à des difficultés techniques et des freinages commerciaux.

Il devient ensuite pendant 5 ans le vice-président pour la recherche et le développement de la « Walt Disney Company » où il a l'occasion de se convaincre chaque jour davantage que, selon ses propres termes, la « révolution informatique ne s'est toujours pas produite » (mais qu'est-ce qui lui faut ?) et que « la meilleure manière de prédire le futur est de l'inventer ». Sa présence auprès de Walt Disney lui permet d'approfondir l'interaction entre les enfants et l'ordinateur, afin de faire de ce dernier une aide véritable à la pédagogie, une stimulation à la créativité et une vraie réponse à la soif insatiable d'apprentissage de l'enfant. Il est aujourd'hui, en compagnie de son ami Nicolas Negroponte du MIT, derrière le projet OLPC « un laptop pour chaque enfant », facilitant par des prix dérisoires (100 $) l'acquisition d'un portable pour chaque enfant. Ces ordinateurs sont largement inspirés de ses premiers travaux sur une interface homme/machine extrêmement conviviale dénommée « dynabook ».

À plus de 60 ans, et de façon à définitivement se donner les moyens de ses ambitions, il fonde sa propre compagnie : « Viewpoints Research Institute », consacrée au développement d'outils informatiques, permettant une meilleure appréhension des systèmes complexes, et destinés tant aux adultes qu'aux enfants. Une des productions de cette nouvelle compagnie est le langage de développement orienté objet, « Squeak », au sujet duquel la maison d'édition Eyrolles a récemment produit un livre[a]. De par sa grande facilité d'utilisation, ce langage devrait permettre tant aux enfants qu'aux éducateurs de « jouer » et « d'expérimenter » de manière créative, ludique et plus intuitive, les maths et la science. Ce nouveau langage, directement issu de Smalltalk, devrait aider Alan Kay à imposer tant ses nouvelles idées sur la pédagogie que sa vision très personnelle, et dont le manque de réceptivité l'obsède, sur l'interaction homme-machine.

Enfin, ne soyez plus étonnés que ce livre reprenne des exemples de biologie et de chimie, nous ne pouvons rêver d'une meilleure compagnie, car Alan Kay obtint son premier diplôme universitaire en biologie moléculaire. C'est à partir de là, et sans arrêt depuis, qu'il se mit à penser l'informatique, son informatique à lui, en des termes biologiques. Ainsi l'idée de messages entre objets est-elle directement inspirée de l'idée de messages chimiques entre les cellules.

a. *Squeak*, Xavier Briffault, Stéphane Ducasse, Eyrolles 2001.

Alan Kay considère qu’un ordinateur idéal se doit de fonctionner comme un organisme vivant, complexe mais se divisant la tâche en un ensemble de modules simples, chaque module devant agir de concert avec les autres (les communications ayant lieu par messages chimiques) afin de réaliser une fonctionnalité commune. Mais tous les modules doivent conserver une certaine autonomie et préserver leur intégrité.

Steve Jobs doit se mordre les doigts de n’avoir saisi qu’une seule innovation importante parmi tout ce dont lui parla Alan Kay, car tout de ce qui fait l’informatique aujourd’hui : les réseaux, la programmation OO, les interfaces graphiques, les imprimantes laser, les ordinateurs de poche, l’informatique ubiquitaire, est redevable en partie à l’extraordinaire imagination et la créativité débordante d’Alan Kay. Allant de déception en déception, à force de voir ses idées spoliées en ce qu’elles ont de plus rémunératrices, il espère par cette nouvelle entreprise arriver enfin, sans doute par sa juste réappropriation et son adoption par les enfants, à redonner toutes ses lettres de noblesse à l’ordinateur, tel qu’il souhaiterait le voir utiliser. Ayant lu tout cela, vous ne serez guère surpris d'apprendre qu'il a reçu en 2004 le prix Turing (la version informatique du Nobel).

Le Paris-Dakar

L’organisateur du Paris-Dakar doit planifier à l’avance la consommation de tout le convoi de véhicules qui participent à cette course : moto, camions, buggy, voiture, hélicoptère… Tous ces véhicules consomment, mais tous le font différemment en fonction des kilomètres parcourus. Il est capital de comprendre ici que c’est la manière de calculer la consommation qui diffère d’un véhicule à l’autre. Si la seule différence se ramenait à une valeur, par exemple, simplement la consommation par km, et que le calcul de la consommation revenait à multiplier cette valeur par le nombre de km à parcourir, il n’y aurait nul besoin d’héritage, nul besoin de polymorphisme. Il n’y aurait, qui plus est, qu’une seule classe de véhicule, dont les objets se distingueraient, notamment par la valeur de cette consommation kilométrique.

Si seule la valeur des attributs différencie les objets entre eux, point n’est besoin de sous-classe. L’usage des langues naturelles, en général, ne permet pas de distinguer le lien existant entre un objet et sa classe d’un côté, et le lien entre une sous-classe et sa superclasse de l’autre. Ma Renault, objet, est une Renault, classe. Une Renault, sous-classe, est une voiture, superclasse. La même expression « est une » est utilisée ici, alors qu’en programmation OO, ces deux contextes d’utilisation mènent à des développements logiciels très différents : simple instanciation d’objet dans le premier, mise en place d’une structure d’héritage à deux niveaux dans le second. Ne vous trompez pas et n’abusez pas d’héritage inutile. Il se doit d’alléger et non pas d’alourdir votre conception. Le gain de son apport, en clarté, économie, maintenance et extensibilité, doit être suffisamment important. Autrement, n’y songez même pas et contentez-vous d’un seul niveau taxonomique. C’est déjà bien assez.

Le polymorphisme en UML

Dans les trois exemples de polymorphisme, présentés plus haut, ce qui apparaît à chaque fois est un type d’architecture logicielle comme celle décrite en UML ci-après.

Dans UML, toute classe ou méthode abstraite est indiquée en italique. Ici, la superclasse et la seule méthode de la classe sont, de fait, abstraites. Trois sous-classes concrètes redéfinissent la méthode `faire-Mon-Numero()`, une méthode dont le corps d’instruction sera radicalement différent pour chacune des sous-classes.

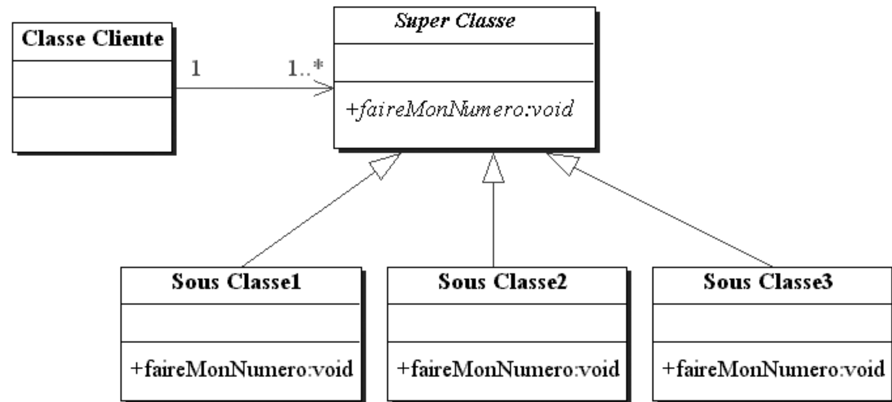


Figure 13-3
Diagramme de classes UML classique associé au polymorphisme.

Une « classe cliente » envoie indifféremment le message `faireMonNumero()` à tous les référents qu'elle possède comme attribut. Comme indiqué dans le diagramme, cet ou ces attributs seront typés par la superclasse. Il peut s'agir d'un et un seul référent, typé statiquement d'une unique manière, mais qui, lors de l'exécution du message, peut endosser plusieurs types dynamiques différents. Cela peut être un tableau de référents, qui lors de l'envoi d'un même message, en boucle sur tous les éléments du tableau, donnera lieu à des exécutions différentes.

Aucun objet n'est ici encore créé, sauf un tableau de référents. Il n'y a donc, à ce stade, pas de tentatives, qui seraient rejetées à la compilation, de création d'objet d'une classe abstraite (la superclasse ici). Les référents pointeront vers des objets, qui eux, lors de leur création, seront de type dynamique, type correspondant à une des sous-classes héritant de la superclasse. L'opération de création d'objets se fera à partir des sous-classes, bien concrètes cette fois. Chaque objet sera donc typé statiquement par son référent superclasse, et typé dynamiquement par sa sous-classe.

Il n'y aurait aucun problème à approfondir l'arbre d'héritage, et à laisser, nonobstant le lien client-serveur entre la classe cliente et la superclasse, la manière dont le message serait exécuté être définie bien plus bas dans l'héritage. Plusieurs couches de classes abstraites peuvent précéder l'arrivée, tout en bas, des classes concrètes. Rappelez-vous qu'à entendre certains gourous de l'OO, les superclasses ne devraient être, en principe, qu'abstraites. Principe discutable, nullement contraint par la syntaxe des langages de programmation, mais dont, néanmoins, on perçoit la pertinence en jetant un simple coup d'œil au monde qui nous entoure.

Exercices

Exercice 13.1

Expliquez pourquoi la présence d'une méthode abstraite dans une classe interdit naturellement la création d'objets issus de cette classe.

Exercice 13.2

Justifiez pourquoi l'absence de concrétisation dans une sous-classe d'une méthode définie abstraite dans sa superclasse oblige, sans autre forme de procédé, la sous-classe à devenir abstraite.

Exercice 13.3

Expliquez pourquoi C++ ne recourt pas à l'utilisation du mot-clé `abstract` pour définir une classe abstraite.

Exercice 13.4

Justifiez pourquoi C# n'impose pas de définir une méthode abstraite `virtual` pour pouvoir la redéfinir.

Exercice 13.5

Réalisez un code dans un des trois langages, dans lequel une superclasse `MoyenDeTransport` contiendrait une méthode `consomme()` abstraite, qu'il faudrait redéfinir dans les trois sous-classes `Voiture`, `Moto`, `Camion`.

Exercice 13.6

Réalisez un code dans un des trois langages, dans lequel une superclasse `ExpressionAlgebrique` contiendrait un `String` comme « 2 + 5 » ou « 7 * 2 » ou « 25 : 5 » et une méthode abstraite `evalue()`, et trois sous-classes `Addition`, `Multiplication`, `Division`, code qui redéfinirait cette méthode selon que l'expression mathématique en question serait une addition, une multiplication ou une division.

Exercice 13.7

Retrouvez ce qu'écrirait à l'écran le code Java suivant. Dessinez également le diagramme de classe UML correspondant.

```
abstract class Militaire {
  private int age;
  private String nationalite;
  private int QI;

  public Militaire(int age, String nationalite, int QI) {
    this.age = age;
    this.nationalite = nationalite;
    this.QI = QI;
  }
  abstract public void partirEnManoeuvre();
  public void deserter() {
    System.out.println("Salut les cocos");
  }
  public void executer() {
    System.out.println("A vos ordres chef");
  }
  public int getQI() {
    return QI;
  }
}
abstract class Plouc extends Militaire {
  public Plouc(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  abstract public void partirEnManoeuvre();
```

```
}
abstract class Grade extends Militaire {
  public Grade(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void commander (Militaire unTroufion) {
    unTroufion.executer();
  }
  abstract public void partirEnManoeuvre();
}
class Colonel extends Grade {
  private Plouc[] mesTroufions;

  public Colonel(int age, String nationalite, int QI, Militaire[] mesTroufions) {
    super(age,nationalite,QI);
    this.mesTroufions = new Plouc[4];
    for (int i=0; i<4; i++) {
      this.mesTroufions[i] = (Plouc)mesTroufions[i];
    }
  }
  public void partirEnManoeuvre() {
    for (int i=0; i<4; i++) {
      commander(mesTroufions[i]);
      System.out.println();
    }
  }
}
class General extends Grade {
  private Colonel monColonel;

  public General(int age, String nationalite, int QI, Colonel monColonel) {
    super(age,nationalite,QI);
    this.monColonel = monColonel;
  }
  public void partirEnManoeuvre() {
    commander(monColonel);
  }
}
class Abruti extends Plouc {
  public Abruti(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void partirEnManoeuvre() {
    System.out.println("C'est super");
  }
}
class TireAuFlanc extends Plouc {
  public TireAuFlanc(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void executer() {
    super.executer();
    deserter();
    System.out.println("et merde");
```

```
  }
  public void partirEnManoeuvre() {
    if (getQI() < 5)
      System.out.println("vivement le bar");
    else
      System.out.println("vivement la bibliotheque");
  }
}
public class Armee {
  public static void main(String[] args) {
    Militaire[] unRegiment = new Militaire[6];
    unRegiment[0] = new Abruti(20, "Belge", 1);
    unRegiment[1] = new Abruti(23, "Francais", 8);
    unRegiment[2] = new TireAuFlanc(20, "Italien", 1);
    unRegiment[3] = new TireAuFlanc(25, "Italien", 8);
    unRegiment[4] = new Colonel(50, "Belge", 2, (Militaire[])unRegiment);
    unRegiment[5] = new General(60, "Francais", 2, (Colonel)unRegiment[4]);
    for (int i=0; i<6; i++)
      unRegiment[i].partirEnManoeuvre();
  }
}
```

Exercice 13.8

Retrouvez ce qu'écrirait à l'écran le code C# suivant :

```
using System;
abstract class Animaux {
  private int monAge;
  private String monNom;

  public Animaux(int age, String nom) {
    monAge = age;
    monNom = nom;
  }
  public virtual void crierSuivantLesAges() {
    if (monAge <= 2) {
      Console.WriteLine("je fais un petit");
    }
    else {
      Console.WriteLine("je fais un gros");
    }
  }
  protected virtual void crierAToutAge() {
    Console.WriteLine("chuuuuut");
  }
  abstract public void dormir();
}
class Fermier {
  private int nbreAnimaux;
  private Animaux[] mesAnimaux;
```

```
  public Fermier(Animaux[] lesAnimaux, int nombre) {
    mesAnimaux = lesAnimaux;
    nbreAnimaux = nombre;
  }
  public void jeFaisMaTourneeDuSoir() {
    for (int i=0; i<nbreAnimaux; i++) {
      mesAnimaux[i].dormir();
      Console.WriteLine();
    }
  }
}
class Cochon : Animaux {
  public Cochon(int age, String nom) : base(age,nom) {}
  protected override void crierAToutAge() {
    Console.WriteLine("groin groin");
  }
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors en fermant les yeux");
  }
}
class Poule : Animaux {
  public Poule(int age, String nom) : base(age,nom) {}
  protected override void crierAToutAge() {
    Console.WriteLine("cot cot");
  }
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors sur mon perchoir");
  }
}
class Taureau : Animaux {
  public Taureau(int age, String nom) : base(age, nom) {}
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors a côte de ma vache");
  }
}
public class Ferme {
  public static void Main() {
    Animaux[] laFamilleRoyale = new Animaux[10];
    Fermier AlphonseII = new Fermier(laFamilleRoyale,3);
    laFamilleRoyale[0] = new Cochon(1,"Phil");
    laFamilleRoyale[1] = new Poule(1,"Astrud");
    laFamilleRoyale[2] = new Taureau(3,"Lorenzio");
    AlphonseII.jeFaisMaTourneeDuSoir();
  }
}
```

Exercice 13.9

Retrouvez ce qu'écrirait à l'écran le code Java suivant. Ce code utilise la classe `Vector` qui est un tableau dynamique dont nous utilisons les méthodes suivantes :

- `size()` pour obtenir la taille du tableau,
- `elementAt(i)` pour extraire le énième élément du tableau (attention, cette méthode renvoie un `object`, et il est nécessaire d'utiliser le `casting` pour récupérer le vrai type),
- `addElement()` rajoute un nouvel élément en queue du tableau.

Dessinez également le diagramme de classe UML correspondant.

```
import java.util.*;
public class UneSerre {
  Vector maSerre = new Vector();
  Jardinier j;

  public static void main(String[] args) {
    new UneSerre();
  }
  public UneSerre() {
    maSerre.addElement(new Bananier(3,150));
    maSerre.addElement(new Olivier(5,300));
    maSerre.addElement(new Magnolia());

    j=new Jardinier(maSerre);

    j.occupeToiDesPlantes(0,2,1);
    j.occupeToiDesPlantes(3,3,4);
    j.occupeToiDesPlantes(4,0,1);
  }
}
class Jardinier {
  Vector maSerre;

  public Jardinier(Vector maSerre) {
    this.maSerre = maSerre;
  }
  public void occupeToiDesPlantes(int periode, int lumiere, int humidite) {
    for (int k=0; k<maSerre.size();k++) {
      Vegetal v = (Vegetal)maSerre.elementAt(k);
      v.jeGrandis(lumiere,humidite,periode);
    }
  }
}
abstract class Vegetal {
  protected int age;
  private int hauteur;
  private int etat;
  protected static String[] humidite=   {   "je me desseche !",
                                            "plus d'eau", "ok pour l'eau",
```

```
                                            "Mes racines pourrissent",
                                            "blou bloup"
                                        };
  protected static String[] lumiere=    {   "more light please",
                                            "lumosite parfaite",
                                            "je suis aveuglee",
                                            "je crame!!"
                                        };
  public Vegetal(int a, int h) {
    age = a;
    hauteur = h;
  }
  abstract public void jeGrandis(int lumiere, int humidite, int periode);
}
abstract class Fruitier extends Vegetal {
  Fruitier(int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFruits() {
    System.out.println(" .... et je donne des fruits");
  }
}
class Bananier extends Vegetal {
  Bananier (int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFruits() {
    System.out.println(" .... et je donne de bonnes bananes");
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("le bananier dit: ");
    if (l>1) l=1;
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==3 && age>3 && age>10) jeDonneDesFruits();
    age++;
  }
}
class Olivier extends Fruitier {
  Olivier(int a, int h) {
    super(a,h);
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("l'olivier dit: ");
    if (l>1) l=1;
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==1 && age>3 && age>10) jeDonneDesFruits();
    age++;
  }
}
```

```
abstract class Plante extends Vegetal {
  Plante(int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFleurs() {
    System.out.println("je donne des jolies fleurs");
  }
}
class Magnolia extends Plante {
  Magnolia() {
    super(0,0);
  }
  Magnolia(int a, int h) {
    super(a,h);
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("le magnolia dit: ");
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==1 && age>1 && age>6) jeDonneDesFleurs();
    age++;
  }
  public void jeDonneDesFleurs() {
    System.out.println("Les Magnolias fleurissent");
  }
}
```

Exercice 13.10

Retrouvez ce qu’écrirait à l’écran le code C++ suivant. Dessinez également le diagramme de classe UML correspondant.

```
#include "stdafx.h"
#include "iostream.h"

class Instrument {
  public:
    Instrument() {}
    virtual void joue() = 0;
};
class Guitare : public Instrument {
  public:
    void joue() {
      cout << "je fais ding ding" << endl;
    }
};
class Trompette : public Instrument {
  public:
    void joue() {
      cout << "je fais pouet pouet" << endl;
    }
```

```
};
class Tambour : public Instrument {
  public:
    void joue() {
      cout << "je fais badaboum" << endl;
    }
};
class Musicien {
  private:
    Instrument* monInstrument;
  public:
    Musicien(Instrument *monInstrument) {
      this->monInstrument = monInstrument;
    }
    void joue() {
      monInstrument->joue();
    }
};
class Orchestre {
  private:
    Musicien *lesMusiciens[3];
    int nombreMusicien;
  public:
    Orchestre(Musicien* lesMusiciens[3]) {
      for (int i=0; i<3; i++) {
        this->lesMusiciens[i] = lesMusiciens[i];
      }
    }
    void joue() {
      for (int i=0; i<3; i++) {
        lesMusiciens[i]->joue();
      }
    }
};
int main() {
  Instrument* lesInstruments[10];
  Musicien* lesMusiciens[8];

  lesInstruments[0] = new Guitare();
  lesInstruments[1] = new Guitare();
  lesInstruments[2] = new Trompette();
  lesInstruments[3] = new Trompette();
  lesInstruments[4] = new Guitare();
  lesInstruments[5] = new Tambour();
  lesInstruments[6] = new Tambour();
  lesInstruments[7] = new Tambour();
  lesInstruments[8] = new Trompette();
  lesInstruments[9] = new Guitare();
  lesMusiciens[0]   = new Musicien(lesInstruments[2]);
```

```
  lesMusiciens[1]   = new Musicien(lesInstruments[5]);
  lesMusiciens[2]   = new Musicien(lesInstruments[0]);
  lesMusiciens[3]   = new Musicien(lesInstruments[9]);
  lesMusiciens[4]   = new Musicien(lesInstruments[9]);
  lesMusiciens[5]   = new Musicien(lesInstruments[4]);
  lesMusiciens[6]   = new Musicien(lesInstruments[2]);
  lesMusiciens[7]   = new Musicien(lesInstruments[1]);

  Musicien* lesMusiciensDOrchestre[3];
  lesMusiciensDOrchestre[0] = lesMusiciens[2];
  lesMusiciensDOrchestre[1] = lesMusiciens[5];
  lesMusiciensDOrchestre[2] = lesMusiciens[3];

  Orchestre *unOrchestre    = new Orchestre(lesMusiciensDOrchestre);
  unOrchestre->joue();

  return 0;
}
```

Exercice 13.11

Corrigez le code des classes A et B pour que le code compile et que son exécution affiche à l’écran : « 1, 2, trois, quatre ». Expliquez et corrigez les erreurs à même le code.

```
abstract class A extends Object {
    private int a,b ;
    private String c ;

    public A(int a,int b, String c) {
       super() ;
       this.a=a ;
       this.b=b ;
       this.c=c ;
    }

    public abstract void decrisToi() ;

}


class B extends A {
   private String d ;

    public B(int a, int b, String c, String d)  {
       this.a=a ;
       this.b=b ;
       this.c=c ;
       this.d=d ;
    }
}
```

```
public class Correction1 {
    public static void main(String[] args)  {

            B b=new B(1,2,"trois","quatre") ;
            b.decrisToi() ;
    }
}
```

Exercice 13.12

Dans le code C++ qui suit, seuls la classe `C` et le `main` contiennent des erreurs. Supprimez-les sans altérer les fonctionnalités du code et indiquez ce que ce code écrirait dans sa version correcte.

```
#include "stdafx.h"
#include "iostream.h"

class A {
private:
        int a ;

public:
        A(int a) {
              this->a=a ;
        }
              int getA() {
              return a ;
        }

        virtual void action(){
              cout << "je travaille" << endl ;
        }
        virtual void actionA() = 0 ;
        void actionA2() {
              cout << "je travaille pour A et A" << endl ;
        }
} ;

class B : A {
private:
        int a,b ;

public:
        B(int a, int b): A(a) {
              this->a=a ;
              this->b = b ;
        }

        void action(){
              actionA() ;
              cout << "je ne travaille pas" << endl ;
        }
              virtual void actionA(){
```

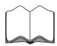
```
                cout << "je travaille pour A" << endl ;
        }
        void actionA2() {
                cout << "je travaille pour A et A" << endl ;
        }
} ;

class C : public A,B {
public:
        C(int a, int b):A(a),B(a,b)
        {}

        void action() {
                cout << getA() << "je travaille pour C"<<endl ;
        }

        void actionA(){
                cout << "je travaille pour C" << endl ;
        }
} ;

int main()
{
        A *a = new A(1) ;
        A *c1 = new C(1,2) ;
        B c2 ;
        c1->action() ;
        c2.action() ;
        return 0 ;
}
```

14

Clonage, comparaison et assignation d’objets

Ce chapitre va nous permettre, par l’entremise de la super-superclasse `Object` en Java, Python et en C# et, différemment en C++ et PHP 5, de comprendre comment l’installation en mémoire des objets et la définition de leurs relations aux autres objets sont déterminantes lors du clonage de ces objets, lors de leur comparaison deux à deux, et lors de l’affectation de l’un d’entre eux à un autre.

Sommaire : La classe Object — Cloner un objet — Le test d’égalité d’objet deux à deux — Affecter un objet à un autre — La surcharge d’opérateur en C++ et en C# — Traitement en surface et traitement en profondeur



Candidus — Jusqu’à quel point peut-on manipuler un objet comme s’il s’agissait d’une simple valeur primitive… un entier par exemple ?

Doctus — La structure d’un objet étant plus complexe, tu devras décider ce qui pourra déterminer que deux objets sont égaux.

Cand. — Qu’une voiture en vaut une autre ou qu’un animal en vaut un autre, par exemple…

Doc. — Tu vois que ça n’a rien d’évident, c’est une question de goûts et de couleurs ! Mais c’est bien par là qu’il faut aborder le problème : les attributs. Il s’agit de bien choisir ceux qu’il faut considérer pour évaluer l’équivalence de deux objets.

Cand. — Je vois : mais quitte à choisir les attributs à comparer deux à deux, pourquoi ne pas les prendre tous ? Et le tour est joué !

Doc. — Mais non, car que feras-tu en présence de deux objets composites ? Te suffira-t-il que deux voitures possèdent un moteur et quatre roues pour les déclarer égales ?

Cand. — Ah ! Je vois le hic. Il faudra donc ouvrir le capot pour voir si un des moteurs n’a pas fait trois tours de compteur alors que l’autre est tout neuf ou que l’un a un turbo et pas l’autre…

Doc. — Autre chose. Lorsque tu vas copier un objet, il te faudra distinguer parfaitement la copie, la représentation de cet objet mémoire étant différente d’un langage à l’autre.

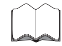
Cand. — Alors là, je sens que ça se complique. Les objets, c’est pas si simple en fait.

Doc. — Non, mais c’est presque simple. Ils savent se cloner. Ils héritent ça de leur grand-mère !

Cand. — D'accord, mais tu me disais qu'il faudra que je m'en mêle pour qu'ils fassent ça comme il faut…

Doc. — Tu devras en effet leur indiquer ce que tu considères équivalent pour guider leur duplication.

Cand. — C'est moi le maître des clones !



Introduction à la classe Object

Si, comme tout membre du règne humain, auquel ils appartiennent malgré leur boulimie de pizza, les informaticiens sont partagés sur l'existence d'une entité spirituelle supérieure, aucun praticien de Java ne doute de l'existence de la super-superclasse, « Ze Klass ». Le sommet de la hiérarchie, la classe des classes, s'appelle en Java, de façon très finaude, la classe `Object`. Les concepteurs de ce langage ont dû trop faire la java la veille du jour décisif, quand il a été décidé du nom de cette classe. Notez dans la série, qu'il existe également une classe `class` en Java dont les objets sont en fait les classes de tous les objets. Insensé non ? Quand on vous disait qu'ils n'étaient pas tout à fait « nets » chez Sun, au point que .Net a décidé de considérablement revoir cette terminologie.

Toutes classes, quelles qu'elles soient, les vôtres comme les nôtres, comme celles de Java, héritent de la classe `Object`. Ce qui nous amène à dire, ce qui pourrait sembler comme la dernière des tautologies, doublée de la première des lapalissades, et pire encore, qu'en Java : tous les objets sont des « Objects ». Il faut croire que ce choix en a convaincu plus d'un sur les vertus de la java, car les concepteurs de C# et Python n'ont pas hésité une seconde pour nommer de façon semblable leur première de la classe. En C#, non seulement toutes les classes, mais également toutes les structures, héritent de la classe `Object`. En Python, il la trouve également suprême, mais pas au point de lui décerner la majuscule (tous ces langages différencient bien évidemment minuscules et majuscules) et cette superclasse a pour petit nom `object`. Elle n'existe ni en C++ ni en PHP 5.

Python et son papa Guido Van Rossum

Python est très souvent plébiscité pour la simplicité de son fonctionnement, ce qui en ferait un langage de choix pour l'apprentissage de la programmation. La programmation est, de fait, très interactive (vous tapez « 1+1 » à l'écran et comme par magie « 2 » apparaît). On arriverait par des écritures plus simples et plus intuitives, donc plus rapidement, au résultat escompté. Si `print "Hello World"` est incontestablement plus simple à écrire que `public static void main (String[] args) {System.out.println "Hello World"}`, nous restons un peu sceptiques quant à l'extension de cette même simplicité à la totalité de la syntaxe. Python reste un langage puissant car il est à la fois OO et procédural. Pour l'essentiel, il n'a rien à envier à des langages comme Java ou C++ et exige donc de maîtriser, comme pour ceux-ci, toutes les bases logiques de la programmation afin de parvenir à des codes d'une certaine sophistication. D'où notre réserve. Il est incontestable, et c'est d'ailleurs la raison de son apparition dans la nouvelle édition de ce livre, que sa popularité n'a cessé de croître au cours des ans, et que de nombreuses solutions logicielles dans le monde Linux ou Microsoft y ont de plus en plus souvent recours. Un Python.Net est sur le point d'éclore.

Ce langage de programmation veut préserver la simplicité qui caractérise les langages de script interprétés plutôt que compilés (les exécutions sont traduites dans un bytecode intermédiaire et s'exécutent au fur et à mesure qu'elles sont rencontrées) grâce à son aspect multi-plates-formes, à l'absence de typage ou à la présence de structures de données très simples d'emploi (listes, dictionnaires et chaînes de caractères). Il souhaite en outre préserver toutes les fonctionnalités qui caractérisent les langages puissants (OO, ramasse-miettes, héritage multiple et librairies d'utilitaires très fournies, comme nous le verrons dans les chapitres qui suivent).

Une telle hybridation, pour autant qu'elle soit réussie, en ferait un langage de tout premier choix pour le prototypage de projet, quitte à revenir par la suite à des langages plus rapides pour la phase finale du projet (comme C++ et Java, avec lequels, d'ailleurs, Python se couple parfaitement : il peut hériter ou spécialiser des classes Java ou C++). Une telle hybridation est-elle possible ? En quoi cette motivation serait-elle innovante par rapport à celle qui a présidé à la création de Java et C#. Ce mélange idéal entre puissance fonctionnelle et simplicité d'usage n'est-il pas le Graal dont tous les langages de programmation d'aujourd'hui sont en quête ?

Python, écrit en C et exécutable sur toutes les plates-formes, est développé par les Python Labs de Zope Corporation qui comprennent une demi-douzaine de développeurs. Ce noyau est dirigé par Guido Van Rossum, inventeur et « superviseur » du langage, qui se targue de porter le titre très ambigu de Dictateur Bénévole à Vie. Toute proposition de modification du langage est débattue par le noyau et soumis à la communauté Python, mais la prise en compte de celle-ci dans le langage reste la prérogative de Guido, qui conserve le dernier mot (d'où son titre : gentil et à l'écoute… mais dictateur tout de même, ces modifications n'étant pas soumises à un vote majoritaire). Guido est hollandais d'origine, détenteur de maîtrises en mathématiques et en informatique de l'université libre d'Amsterdam. Il participa, à la fin des années 1980, à un groupe de développeurs dont le but était la mise au point d'un langage de programmation abordable par des non-experts, d'où son nom « ABC ». Dès 1991, il s'attaque à Python. (Ce nom ne doit rien à l'horrible reptile mais se réfère aux extraordinaires humoristes anglais que sont les Monthy Python, qui ont révolutionné dans les années 1970 et 1980 et l'humour et la télévision, et dont l'humour nous manque tellement aujourd'hui, d'où la nécessité de les remplacer par un jumeau ouvert sur le Web et la programmation.) Guido Van Rossum travaille alors au CWI (Centrum voor Wiskunde en Informatica). En 1995, il prend la direction des États-Unis, comme tout bon informaticien qui se respecte et qui veut percer, et travaille pour le CNRI (Corporation for National Research Initiatives) jusqu'en 2000. À cette époque, il publie *Computer Programming for Everybody*, sa profession de foi pour l'enseignement de la programmation. C'est également à cette époque qu'il est nommé directeur des Python Labs de Zope Corporation. Depuis 2005, il travaille pour Google, entreprise connue pour son immense succès commercial, et qui semble investir beaucoup dans le langage Python.

La communauté Python reste très structurée autour d'un seul site Web : http://www.python.org, ce qui s'avère un atout considérable, surtout lors de l'apprentissage et de la découverte du langage. À quelques subilités près, Python est dans le prolongement de l'aventure Open Source, dont le représentant le plus emblématique reste Linux. Il pourrait devenir le langage de programmation phare de l'Open Source, tout comme Linux est celui de l'OS. Toutes les sources restent accessibles et disponibles, monsieur-tout-le-monde peut participer à l'évolution du produit, mais la prise en charge officielle de ces évolutions reste sous la responsabilité de quelques-uns (ici, un seul). Il semble que les éditeurs informatiques, tant Sun que Microsoft, évoluent pour leur bébé, vers ce même modèle de développement. Java est en effet devenu Open Source et nous avons eu l'occasion d'évoquer le Projet Mono, version Open Source de .Net.

Quant aux références pour aborder ce langage, le site officiel de Python reste une source idéale. Des livres, essentiellement publiés par les éditions O'Reilly, sont disponibles, parmi lesquels :

– *Apprendre à programmer avec Python*, de GÉRARD SWINNEN : très introductif et très axé procédural ;
– *Programing Python*, de TAILL : pour une présentation très complète ;
– *Python en concentré – Manuel de référence* (ce qu'il fut indéniablement pour l'écriture de notre livre), d'ALEX MARTELLI.

Une classe à compétence universelle

Une telle classe présente ce premier avantage que vous pouvez l'utiliser comme argument ou type de retour d'une méthode, que vous souhaitez à compétence universelle (pouvant s'opérer sur toute sorte d'objet), méthode que vous pourrez, par la suite, spécialiser selon le type d'objet en question. Cette classe `Object` est donc, le plus souvent, candidate à une utilisation de type « universelle ». Vous pouvez l'utiliser quand vous concevez un type de structure particulière, qui concerne tous les objets sans distinction de classe, comme une liste liée ou un tableau extensible.

En Java et C #, toute une panoplie de classes `collections` fait largement usage de la classe `Object`. Les exemples les plus connus en sont les classes `Vector` et `ArrrayList`, tableaux extensibles, qui peuvent contenir un nombre indéterminé d'objets de toute classe. Par exemple, les méthodes de ce `Vector` les plus usitées sont `addElement(Object unObjet)`, qui permet de rajouter n'importe quel type d'objet à la fin de ce `Vector`, et `Object elementAt(int i)`, méthode qui renvoie l'objet positionné à la « énième position » du `Vector`.

Comme vous le voyez, c'est bien le type `Object` que l'on retrouve dans la définition de ces méthodes. C'est normal, car rien dans la fonctionnalité de ce `Vector` n'exige de connaître le type particulier de ce qui y est

contenu. Les `Vectors` étant généralement composés d’objets de classe quelconque, l’utilisation de la méthode `elementAt(int i)` est, presque toujours, accompagnée d’un « casting », afin de récupérer les caractéristiques qui sont propres à l’objet extrait du `Vector`. Un exemple d’utilisation de la classe `Vector` est donné ci-après. Nous y découvrons également pourquoi et comment la possibilité, depuis les dernières versions de Java et de .Net, de typer ces collections, permet d’éviter l’utilisation (fort décriée nous l’avons vu précédemment) du « casting ». Comme nous le verrons aussi, les objets se nomment et se clonent tous, mais tous peuvent le faire d’une manière qui leur est particulière.

Code Java illustrant l’utilisation de la classe Vector et innovation de Java 5

```
import java.util.*;
class O1 {
  public void jeTravaillePourO1() {
    System.out.println("Salut, je travaille pour O1");
  }
}
class O2 {
  public void jeTravaillePourO2() {
    System.out.println("Salut, je travaille pour O2");
  }
}
public class TestVector {
  public static void main(String[] args) {
    Vector unVecteur = new Vector() ; /* dans le nouveau Java, vous pouvez créer :
        ➥Vector<O1> unVecteur = new Vector<O1> */
    unVecteur.addElement(new O2());
    if (unVecteur.elementAt(0) instanceof O1)
      ((O1)unVecteur.elementAt(0)).jeTravaillePourO1(); /* il faut " caster " pour récupérer
      ➥le bon type, mais plus dans la version nouvelle du langage */
    if (unVecteur.elementAt(1) instanceof O2)
      ((O2)unVecteur.elementAt(1)).jeTravaillePourO2();
  }
}
```

Résultat

```
Salut, je travaille pour O1
Salut, je travaille pour O2
```

Nouvelle version du code

```
import java.util.*;
class O1 {
  public void jeTravaillePourO1() {
    System.out.println("Salut, je travaille pour O1");
  }
}
```

```
class O2 {
  public void jeTravaillePourO2() {
    System.out.println("Salut, je travaille pour O2");
  }
}
public class TestVector2 {
  public static void main(String[] args) {
    Vector<O1> unVecteur = new Vector<O1>(); // depuis le nouveau Java
    unVecteur.add(new O1());
 // « unVecteur.add(new O2()); » n'est plus possible
    unVecteur.elementAt(0).jeTravaillePourO1(); // Plus besoin de caster !!!!
  }
}
```

Bien sûr, s'il n'est plus besoin de caster, il devient impossible d'utiliser ce même vecteur pour des objets de type différent (sauf si issus d'une même superclasse). Dans notre exemple, il n'est plus possible d'insérer des objets de la classe `O1` et `O2` dans le même `Vector`.

Décortiquons la classe Object

Voici maintenant la classe `Object`, telle qu'elle est spécifiée par Sun. Onze méthodes y sont prédéfinies, représentatives de la compétence universelle de chaque objet en Java.

```
public class Object {
  private static native void registerNatives();
  static {
    registerNatives();
  }
  public final native Class getClass();
  public native int hashCode();
  public boolean equals(Object obj) {
    return (this == obj);
  }
  protected native Object clone() throws CloneNotSupportedException;
  public String toString() {
    return getClass().getName() + "@" +   Integer.toHexString(hashCode());
  }
  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long timeout) throws InterruptedException;
  public final void wait(long timeout, int nanos) throws InterruptedException {
    if (timeout < 0)
      throw new IllegalArgumentException("timeout value is negative");
    if (nanos < 0 || nanos > 999999)
      throw new IllegalArgumentException(
          "nanosecond timeout value out of range");
    if (nanos >= 500000 || (nanos != 0 && timeout == 0))
      timeout++;
    wait(timeout);
  }
```

```
  public final void wait() throws InterruptedException {
    wait(0);
  }
  protected void finalize() throws Throwable { }
}
```

Décrire chacune de ces méthodes dépasserait largement le cadre de ce voyage initiatique dans le monde de l’OO ; nous devrions rentrer trop profondément dans des arcanes syntaxiques propres à Java. Ainsi, la présence du mot `native` dans la déclaration des méthodes signale que celles-ci sont écrites dans un autre langage de programmation, pour des raisons d’optimisation ou de proximité intime avec le fonctionnement du processeur, généralement C ou C++. Aucune instruction n’est donc présente, et la version exécutable de la méthode est déjà pré-installée dans la machine virtuelle Java, dédiée à la plate-forme que vous utilisez. De même, les méthodes `notify` et `wait` jouent un rôle clé lors de la mise en pratique du multithreading, que nous introduirons au chapitre 17.

Notez finalement (et nous l’illustrerons par la suite) que les deux méthodes dont l’encapsulation est du type controversé `protected` sont précisément celles qui sont les plus susceptibles de subir une redéfinition dans les sous-classes, redéfinition faisant appel à la version d’origine (d’où le `protected`). En cela, elles font partie du SPM (Société pour la Protection des Méthodes).

Penchons-nous plutôt sur ces méthodes qui aident à la compréhension des mécanismes OO, car il est intéressant de voir ce que les brillants ingénieurs de SUN considèrent comme étant des méthodes à caractère universel, méthodes pouvant être exécutées sur tous les objets informatiques qui peuplent notre galaxie. La preuve en est que certaines de ces méthodes se retrouvent, au nom près, dans la classe `Object` du langage C#. Nous les indiquons ci-après :

```
public static bool Equals  (Object objA , Object objB)
public virtual bool Equals (Object obj)
public virtual int GetHashCode ()
public Type GetType()
public static bool ReferenceEquals(Object objA, Object objB)
public virtual string ToString()
protected object MemberwiseClone()
```

Nous avons déjà vu la méthode `ToString()`, qui permet à l’objet de se présenter, tout en nous renseignant sur sa classe. Des méthodes comme `GetType()` en C# ou `GetClass()` en Java permettent également un semblant d’introspection où l’objet, lui-même, peut informer celui qui le manipule sur la nature de sa classe et, par là même, obtenir toutes les informations désirées sur les méthodes ou les attributs qui caractérisent cette classe. Ainsi, il est possible de créer un nouvel objet `unAutreO` à partir d’un objet existant, `unO`, au moyen de la simple instruction :

```
Object unAutreO = unO.getClass().newInstance(); // En Java
```

La classe `object` en Python, dont il faut, au contraire de Java et C#, explicitement hériter si l’on souhaite récupérer certaines fonctionnalités, se caractérise également par un ensemble de méthodes à compétence universelle comme `__init__` pour construire n’importe quel objet et `__new__` pour le construire à partir d’un objet existant. Nous avons déjà vu la méthode `__str__` qui permet à l’objet un début d’introspection et de nous renseigner à l’exécution sur son typage dynamique. Toute sous-classe de `object` peut redéfinir ces méthodes.

Mais revenons à la classe `Object` de Java et surtout à deux méthodes universelles qui nous intéressent plus particulièrement ; tout d’abord, la méthode : `equals(Object o)`, qui sert à tester l’égalité de deux objets. Elle est appelée sur le premier objet, en Java, afin de le comparer au second. Elle peut être appelée de la même

manière en C#, ou appelée en lui passant en argument les deux objets à comparer (dans ce dernier cas, elle devient légitimement `static`). La seconde méthode est `clone()` qui, en Java, permet de dupliquer un objet. Par les petits codes suivants, nous allons illustrer au mieux le fonctionnement de ces deux méthodes. Notre compréhension des problèmes de stockage et d'organisation en mémoire tas des objets devrait s'en trouver améliorée. Nous allons étudier en premier lieu la méthode `equals(Object o)`.

Test d'égalité de deux objets

Code Java pour expérimenter la méthode equals(Object o)

Dans le premier programme qui suit, nous codons nos habituelles classes `O1` et `O2`. La classe `O2` possède juste un attribut entier, alors que la classe `O1` possède un attribut entier et un attribut de type référent vers `O2`. Nous créons ensuite trois objets `O2` et deux objets `O1` dont nous testerons l'égalité. Dans un premier temps, plusieurs instructions seront mises en commentaire afin de les désactiver.

```
class O1{
  private int unAttributO1;
  private O2 lienO2;

  public O1(int unAttributO1, O2 lienO2){
    this.unAttributO1 = unAttributO1;
    this.lienO2 = lienO2;
  }
  public boolean equals(Object unObjet) /* la méthode qui nous intéresse, d'abord désactivée
  ➥puis activée */ {
    if (this == unObjet) {
      return true;  /* renvoie true si les objets sont les mêmes */
    }
    else {
      if (unObjet instanceof O1) {
        O1 unAutreO1 = (O1)unObjet;  /* effectue un " casting " */
        if ((unAttributO1 == unAutreO1.unAttributO1)
            &&(lienO2.getAttribut() == unAutreO1.lienO2.getAttribut())){
          return true;
        }
        else{
          return false;
        }
      }
    }
    return false;
  }
}
class O2{
  private int unAttributO2;

  public O2(int unAttributO2){
    this.unAttributO2 = unAttributO2;
  }
```

```
  public int getAttribut(){
    return unAttribut02;
  }
/*
  public boolean equals(Object unObjet) {
    if (this == unObjet) {
      return true;
    }
    else {
      if (unObjet instanceof O2) {
        O2 unAutreO2 = (O2)unObjet;
        if (unAttribut02 == unAutreO2.unAttribut02)
          return true;
        else
          return false;
      }
    }
    return false;
  }
*/
}
public class CloneEqual{
  public static void main(String[] args){
    O2 unO2 = new O2(5);
    O2 unAutreO2 = new O2(5);
    O2 unTroisièmeO2 = new O2(10);
    unTroisièmeO2 = unO2;
    O1 unO1 = new O1(10, unO2);
    O1 unAutreO1 = new O1(10, unAutreO2);

    if (unO2 == unAutreO2) /* teste l'égalité des référents */
      System.out.println("unO2 et unAutreO2 ont la même référence");
    if (unO2.equals(unAutreO2)) /* sans redéfinition, teste l'égalité des référents, avec
    ➥redéfinition teste l'égalité des états */
      System.out.println("unO2 et unAutreO2 ont le même état") ;
    if (unO2 == unTroisièmeO2)
      System.out.println("unO2 et unTroisièmeO2 ont la même référence");
    if (unO2.equals(unTroisièmeO2))
      System.out.println("unO2 et unTroisièmeO2 ont le même état");
    if (unO1 == unAutreO1)
      System.out.println("unO1 et unAutreO1 ont la même référence");
    if (unO1.equals(unAutreO1))
      System.out.println("unO1 et unAutreO1 ont le même état");
  }
}
```

Nous avons, dans un premier temps, désactivé la redéfinition de la méthode `equals` dans les classes `O1` et `O2`. Voici le résultat du code, en l'absence de cette redéfinition :

Résultat

```
un02 et unTroisième02 ont la même référence
un02 et unTroisième02 ont le même état
```

On comprend, au vu de ce résultat, le mode de fonctionnement par défaut de la méthode `equals()`, celle qui est héritée de la classe `Object`. En fait, cette méthode, sans redéfinition, se comporte exactement comme le double « = », opération d'égalité logique venant du C++ (à ne pas confondre avec le simple « = », qui est l'opération d'affectation). Par défaut, l'égalité porte sur les référents, c'est-à-dire les adresses des objets. Il est clair que, si les référents sont égaux, on sait que l'on pointe vers un seul et même objet, et donc l'égalité est tout à fait vérifiée. Ce test d'égalité des référents est précieux, quand la complexité du programme devient telle qu'il est nécessaire, à certaines étapes du code, de vérifier que deux référents continuent à pointer vers un même objet. Cela se produit très souvent quand les référents sont perdus dans d'immenses vecteurs ou tableaux.

Ce que l'on aimerait pourtant, comme illustré dans la figure qui suit, c'est élargir cette égalité aux objets qui, bien qu'installés dans des zones mémoire différentes, possèdent un même état, c'est-à-dire des attributs ayant la même valeur. Lorsque le programme écrit « `un02` et `unTroisième02` ont le même état », il a raison, mais cela n'apprend rien, puisqu'il s'agit du même objet en mémoire. Or, deux objets d'une même classe, et caractérisés par un même état, pourraient légitimement être considérés comme égaux, où qu'ils se trouvent dans la mémoire.

Vous pourriez, dès lors, conserver le double « = » pour le test d'égalité des référents, et redéfinir la méthode `equals()` pour le test d'égalité des états. Car c'est parce que Java sait que vous aurez tôt ou tard besoin ou envie de redéfinir cette nouvelle procédure de comparaison qu'il a créé et installé la méthode `equals()` dans la classe des classes, méthode qui ne demande qu'à être redéfinie, comme suit :

```
public boolean equals(Object unObjet) {
  if (this == unObjet) {
    return true;
  }
  else {
    if (unObjet instanceof O2) {
      O2 unAutreO2 = (O2)unObjet;
      if (unAttributO2 == unAutreO2.unAttributO2)
        return true;
      else
        return false;
    }
  }
  return false;
}
```

D'abord, on teste s'il s'agit oui ou non du même objet. Si ce n'est pas le cas, on teste si ces deux objets sont bien issus de la même classe. Enfin, dans le cas positif, on compare les valeurs des attributs deux à deux.

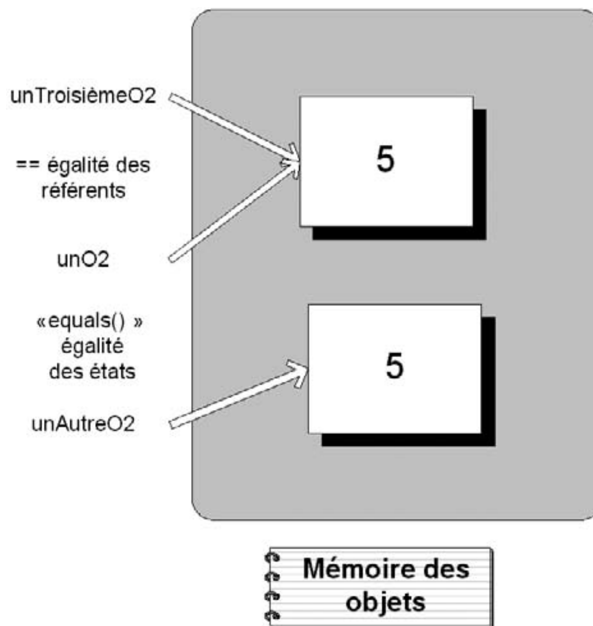


Figure 14-1
Différence entre l'égalité des référents et l'égalité des états.

Cette redéfinition des deux méthodes `equals()` dans les deux classes était déjà présente en commentaire. Si vous supprimez les marques de commentaire et ré-exécutez le programme vous obtenez :

Résultat

```
unO2 et unAutreO2 ont le même état
unO2 et unTroisièmeO2 ont la même référence
unO2 et unTroisièmeO2 ont le même état
unO1 et unAutreO1 ont le même état
```

Égalité en profondeur

On s'aperçoit que, bien qu'`unO2` et `unAutreO2` ne représentent plus physiquement le même objet, ils sont malgré tout déclarés comme égaux, car ils partagent les mêmes valeurs d'attributs. Quitte à redéfinir la méthode `equals()`, il est important de la redéfinir le plus « profondément » possible. En effet, deux objets seront égaux, si, avant tout, ils possèdent les mêmes valeurs attributs. Cependant, comme vous le montre l'exemple de la classe `O1` et la figure qui suit, il faut que les objets qu'ils réfèrent par leur attribut de type « référent », possèdent, à leur tour, les mêmes valeurs d'attributs, et ainsi de suite, de référents en référents. Cette procédure de comparaison doit donc s'effectuer récursivement, en suivant le fil rouge des référents, et en parcourant tout le réseau relationnel des objets.

Il ne faut pas seulement que les attributs soient égaux, au premier niveau, il faut également que les objets vers lesquels pointent les attributs référents soient eux-mêmes égaux. La redéfinition de la méthode `equals()` dans le code de la classe `O1` illustre ce mécanisme de comparaison en cascade. Et c'est ainsi que, dans le résultat de l'exécution du programme, les objets `unO1` et `unAutreO1` sont également déclarés égaux.

Parler de récursivité est parfaitement adéquat, car une manière plus élégante et vraiment récursive de redéfinir la méthode `equals` dans la classe `O1` aurait été :

```
public boolean equals(Object unObjet)  /*la méthode qui nous intéresse, d'abord désactivée
  puis activée*/  {
    if (this == unObjet) {
      return true;   //renvoie true si les objets sont les mêmes
    }
    else {
      if (unObjet instanceof O1) {
        O1 unAutreO1 = (O1)unObjet;   //effectue un " casting "
        if ((unAttributO1 == unAutreO1.unAttributO1)
            &&(lienO2.equals(unAutreO1.lienO2))){ // appel vraiment récursif de "equals"
          return true;
        }
        else{
          return false;
        }
      }
    }
    return false;
```

Les bonnes utilisation et redéfinition de cette méthode sont conditionnées par une compréhension adéquate des modes d'adressages et de stockage des objets en mémoire. Il en va de même de la méthode `clone()`, permettant de dupliquer un objet, et que nous illustrons en enrichissant le code précédent, par la possibilité de cloner les objets `O1` et `O2`.

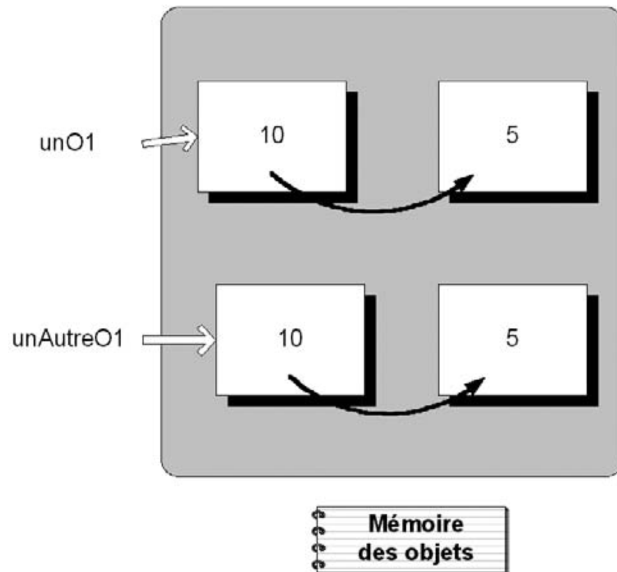


Figure 14-2

Pour que deux objets soient égaux, il faut non seulement que leurs attributs soient égaux, mais que les objets vers lesquels ils pointent soient égaux également.

Le clonage d'objets

Code Java pour expérimenter la méthode clone()

```
class O1 implements Cloneable { /* implémenter l'interface Cloneable */
  private int unAttributO1;
  private O2 lienO2;  /* l'attribut référent */

  public O1(int unAttributO1, O2 lienO2) {
    this.unAttributO1 = unAttributO1;
    this.lienO2 = lienO2;
  }
  public void donneAttribut() {
    System.out.println("valeur attribut = " + unAttributO1);
  }
  public O2 getO2() {
    return lienO2;
  }
  public boolean equals(Object unObjet) {
    if (this == unObjet) {
      return true;
    }
    else {
      if (unObjet instanceof O1) {
        O1 unAutreO1 = (O1)unObjet;
        if ((unAttributO1 == unAutreO1.unAttributO1)
            &&(lienO2.getAttribut()== unAutreO1.lienO2.getAttribut())) {
          return true;
        }
        else {
          return false;
        }
      }
    }
    return false;
  }
  public Object clone() throws CloneNotSupportedException { /* la méthode clone */
    O1 unNouveauO1      = (O1)super.clone() ; /* copie superficielle et rappel de la version d'origine */
    unNouveauO1.lienO2 = (O2)lienO2.clone(); /* copie en profondeur */
    return unNouveauO1;
  }
}
class O2 implements Cloneable {
  private int unAttributO2;

  public O2(int unAttributO2) {
    this.unAttributO2 = unAttributO2;
  }
  public int getAttribut() {
    return unAttributO2;
  }
```

```
  public void setAttribut(int unAttribut02) {
    this.unAttribut02 = unAttribut02 ;
  }
  public boolean equals(Object unObjet) {
    if (this == unObjet) {
      return true;
    }
    else {
      if (unObjet instanceof 02) {
        02 unAutre02 = (02)unObjet;
        if (unAttribut02 == unAutre02.unAttribut02)
          return true;
        else
          return false;
      }
    }
    return false;
  }
  public Object clone() throws CloneNotSupportedException { /* la méthode clone */
    return super.clone();
  }
}
public class CloneEqual {
  public static void main(String[] args) {
    /*   Test de la méthode equal()    */
    02 un02 = new 02(5);
    02 unAutre02 = new 02(5);
    02 unTroisieme02 = new 02(10) ;
    01 un01 = new 01(10, un02);
    01 unAutre01 = new 01(10, unAutre02);

    if (un02 == unAutre02)
      System.out.println("un02 et unAutre02 ont la même référence");
    if (un02.equals(unAutre02))
      System.out.println("un02 et unAutre02 ont le même état");
    if (un02 == unTroisieme02)
      System.out.println("un02 et unTroisieme02 ont la même référence") ;
    if (un02.equals(unTroisieme02))
      System.out.println("un02 et unTroisieme02 ont le même état ")  ;
    if (un01 == unAutre01)
      System.out.println("un01 et unAutre01 ont la même référence");
    if (un01.equals(unAutre01))
      System.out.println("un01 et unAutre01 ont le même état");

    /* Test de la méthode Clone  */
    02 unQuatrieme02 = null ;
    try {
      unQuatrieme02 = (02)unTroisieme02.clone() ; /* clonage */
    } catch(Exception e) {}

    /* vérification de l'égalite de " unTroisieme02 " et " unQuatrieme02 " */
```

```
    System.out.println(unTroisieme02.getAttribut() + " = ? " + unQuatrieme02.getAttribut()) ;
    01 unTroisieme01 = null ;
    try {
      unTroisieme01 = (01)unAutre01.clone() ;
    } catch(Exception e) {}

    if (un01.equals(unTroisieme01))
      System.out.println("un01 et unTroisieme01 ont le même état") ;
    unTroisieme01.get02().setAttribut(7) ;

    /* on modifie l'état de l'objet "unTroisieme01" et on vérifie que cela n'affecte pas l'objet un 01 */
    if (un01.equals(unTroisieme01))
      System.out.println("un01 et unTroisieme01 ont le même état") ;
    else
      System.out.println("un01 et unTroisieme01 n'ont pas le même état") ;
  }
}
```

Résultat

```
un02 et unAutre02 ont le même état
un01 et unAutre01 ont le même état
10 = ? 10
un01 et unTroisieme01 ont le même état
un01 et unTroisieme01 n'ont pas le même état
```

Java nécessite quelques additions syntaxiques pour pouvoir cloner un objet. D'abord, il faut, par l'implémentation d'une interface particulière (le chapitre 15 sera entièrement consacré à l'implémentation d'interfaces), déclarer que les deux classes `01` et `02` peuvent être clonées. Cette addition est une simple étiquette apposée aux deux classes, nécessaire lors de l'exécution du clonage, réalisée par une méthode `native` en Java, donc pouvant poser problème. En effet, le clonage pouvant échouer et lever, comme vous le constatez dans la signature de la méthode `clone()`, une exception, Java oblige à prévoir cette exception, que vous devenez contraints et forcés de gérer, comme toute exception. Une fois ces additions effectuées, l'appel de la méthode `clone()` de la classe `Object` a pour effet de créer une copie de l'objet, attribut par attribut.

Cela ne pose aucun problème, comme le résultat du code l'indique, pour la classe `02`, car il suffit de dupliquer la valeur du seul attribut qu'elle possède, et de l'installer dans le clone. La redéfinition de `clone()` dans la classe `02` se limite, de fait, à rappeler la version de la classe `Object`. Mais comme cette méthode a un accès `protected` dans la classe `Object`, vous être forcés de la redéfinir en la déclarant `public` dans la classe `02` de manière à pouvoir l'utiliser (souvenez-vous que l'encapsulation ne peut que s'affaiblir en restriction, lors d'une redéfinition des méthodes dans les sous-classes). La présence de ce `protected` est de nouveau une incitation de Java à maîtriser au mieux l'utilisation du clonage par un rappel de la méthode d'origine. En effet, en matière de clonage, vous n'êtes pas à l'abri d'une surprise, et vous pourriez vous retrouver avec un petit objet aussi inattendu que Doly…

Surprise il peut en effet y avoir, car la situation est de nouveau plus délicate pour `01`, étant donné que deux solutions s'offrent à vous, comme l'illustre la figure suivante. Soit vous optez pour une copie superficielle de tous les attributs de l'objet `01` dans un nouvel objet, appelé dans le code `unTroisième01`. Dans ce cas, et en ce qui concerne l'attribut de `01` référent vers l'objet `02`, le nouvel objet `01`, clone du premier, partagera la valeur

de cet attribut et, simplement, se mettra également à référer le même objet O2. Mais il peut sembler préférable qu'à l'instar du clonage de l'objet O1, vous cloniez également tous les attributs référés par cet objet.

De nouveau, le clonage pourrait se propager récursivement de référent à référent, de manière à reproduire, à partir d'un premier objet, tout le réseau relationnel dans lequel il s'inscrit. C'est l'option prise par le code ici, qui rajoute comme attribut référent du nouvel objet O1, un clone de cet attribut. De manière à illustrer ce mécanisme de clonage en profondeur, à la fin du programme, on modifie l'attribut de l'objet O2 pointé vers l'objet `unTroisièmeO1`. Si l'objet O2 était pointé deux fois par les deux objets O1, le résultat du test de comparaison serait différent. Il y a donc bien deux objets O1 et deux objets O2 distincts.

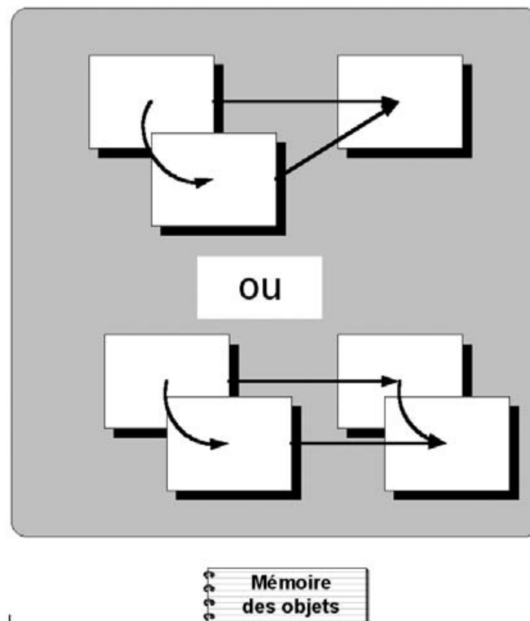


Figure 14-3
Différence entre clonage en superficie et clonage en profondeur.

Comme nous l'avions déjà constaté lors de l'étude de la méthode `equals()`, et à nouveau ici pour le clonage, il y a plusieurs options dans la manipulation des objets, selon que l'on entraîne dans ces mêmes manipulations les objets référés ou pas. Ces deux méthodes `equals` et `clone` peuvent se prêter semblablement à une redéfinition récursive. Il sera toujours indispensable de maîtriser les conséquences que ces choix entraînent, bien que Java et C# (qui, dans les opérations que nous avons effectuées ici, lui ressemble comme deux gouttes d'eau) vous forcent la main et vous épaulent largement pendant ces manipulations. Ainsi, le « ramasse-miettes » pourra vous débarrasser d'objets maladroitement créés lors de ces manipulations.

Égalité et clonage d'objets en Python

Code Python pour expérimenter l'égalité et le clonage

Dans le code Python qui suit, la pratique de l'égalité d'objets est singulière et passe forcément par l'utilisation de l'opérateur ==. Toutefois, à vous de décider quel type d'égalité vous choisissez de réaliser et cela pour chaque

classe. Cela se fait par la définition de la méthode __eq__, qui sera automatiquement et implicitement appelée dans l'exécution du code dès que l'opération == entre deux objets est rencontrée. Il en va de même pour les méthodes __ge__, __gr__, __le__, __lt__ et __ne__, automatiquement appelées dès que sont rencontrés respectivement les opérateurs : >=, >, <=, < et !=.

Nous avons, dans ce code, décidé de réaliser la version de « l'égalité d'état en profondeur ». En l'absence de définition de la méthode __eq__, la version par défaut est, comme en Java, celle de l'égalité des référents. En revanche, aucune fonctionnalité ne vous mâche la besogne pour le clonage d'objets et nous nous sommes limités à définir de toutes pièces une méthode clone (uniquement dans la classe O2), qui renvoie une nouvelle instance avec l'attribut de l'instance que l'on choisit de cloner.

```
  class O1:
    __unAttributO1 = 0
    __lienO2 = None

    def __init__(self,unAttributO1,lienO2):
       self.__unAttributO1 = unAttributO1
       self.__lienO2 = lienO2

    def __eq__(self, unObjet):
        if isinstance(unObjet,O1) :
           if (self.__unAttributO1 == unObjet.__unAttributO1 and
               self.__lienO2.getAttribut() == unObjet.__lienO2.getAttribut()):
                return True
           else:
                return False
        else:
          return False


class O2:
    __unAttributO2 = 0

    def __init__(self,unAttributO2):
        self.__unAttributO2 = unAttributO2

    def __eq__(self, unObjet):
        if isinstance(unObjet,O2):
            if (self.__unAttributO2 == unObjet.__unAttributO2):
                return True
            else:
                return False
        else:
            return False
    def clone(self):
      return O2(self.__unAttributO2)

    def getAttribut(self):
      return self.__unAttributO2

unO2 = O2(5)
```

```
unAutreO2 = O2(5)
unTroisiemeO2 = O2(10)
unTroisiemeO2 = unO2
unO1 = O1(10, unO2)
unAutreO1 = O1(10,unAutreO2)

if unO2 == unAutreO2:
    print "unO2 et unAutreO2 ont le meme etat"
if unO2 == unTroisiemeO2:
    print "unO2 et unTroisiemeO2 ont le meme etat"
if unO1 == unAutreO1:
    print "unO1 et unAutreO1 ont le meme etat"

unQuatreO2 = unO2.clone()
print unAutreO2.getAttribut()
```

Résultats

```
unO2 et unAutreO2 ont le même état
unO2 et unTroisièmeO2 ont le même état
unO1 et unAutreO1 ont le même état
5
```

Égalité et clonage d’objets en PHP 5

Code PHP 5 pour expérimenter l’égalité et le clonage

Dans l’esprit, PHP 5 qui suit est très proche des codes qui précèdent mais à nouveau en présence d’une syntaxe considérablement modifiée. Il est tout d’abord nécessaire de différencier l’égalité des référents, assurée avec un `===` (l’inégalité avec `!==`) de l’égalité des états des objets (c’est-à-dire des attributs), égalité qui sera réalisée de manière récursive (et ceci par défaut) et en présence d’un « `==` », l’inégalité avec `!=`. C’est donc bien le nombre de « = » qui fait la différence. En ce qui concerne le clonage, il en existe par défaut une forme de clonage implicite (appelé par la syntaxe `clone $object`) qui recopie les attributs un à un dans le nouvel objet, mais qu’il est possible de redéfinir par la définition d’une fonction `__clone()` dans la classe concernée. Si celle-ci existe, c’est elle qui sera appelée lors du clônage de l’objet, comme le code qui suit l’illustre au mieux.

```
<html>
<head>
<title> Clonage et comparaison d'objets </title>
</head>
<body>
<h1> Clonage et comparaison d'objets </h1>
<br>

<?php
     class O1 {
          private $unAttributO1;
          private $lienO2;
```

```
    public function __construct($unAttribut01, $lien02) {
        $this->unAttribut01 = $unAttribut01;
        $this->lien02 = $lien02;
    }

    public function donneAttribut() {
        print ("valeur attribut = $this->unAttribut01 <br> \n");
    }

    public function get02() {
        return $this->lien02;
    }

    public function __clone() {
        $this->lien02 = clone $this->lien02;
    }
}

class 02 {
    private $unAttribut02;

    public function __construct($unAttribut02) {
        $this->unAttribut02 = $unAttribut02;
    }

    public function getAttribut() {
        return $this->unAttribut02;
    }

    public function setAttribut($unAttribut02) {
        $this->unAttribut02 = $unAttribut02;
    }
}

$un02 = new 02(5);
$unAutre02 = new 02(5);
$unTroisieme02 = new 02(10);
$un01 = new 01(10, $un02);
$unAutre01 = new 01(10, $unAutre02);

if ($un02 === $unAutre02) {
    print ("un02 et unAutre02 ont la même référence<br> \n");
}
if ($un02 == $unAutre02) {
    print ("un02 et unAutre02 ont le même état <br> \n");
}
if ($un02 === $unTroisieme02) {
    print ("un02 et unTroisieme02 ont la même référence <br> \n");
}
if ($un02 == $unTroisieme02) {
    print ("un02 et unTroisieme02 ont le même état <br> \n");
}
if ($un01 === $unAutre01) {
    print ("un01 et unAutre01 ont la même référence <br> \n");
}
```

```
        if ($un01 == $unAutre01) {
             print ("un01 et unAutre01 ont le même état <br>\n");
        }

        $unQuatrieme02 = clone $unTroisieme02;
        print ($unTroisieme02->getAttribut());
        print ($unQuatrieme02->getAttribut());
        $unTroisieme01 = clone $unAutre01;
        if ($un01 == $unTroisieme01) {
             print ("un01 et unTroisieme01 ont le même état <br>\n");
        }

        $unTroisieme01->get02()->setAttribut(7);

        if ($un01 == $unTroisieme01) {
             print ("un01 et unTroisieme01 ont le même état <br>\n");
        } else {
             print ("un01 et unTroisieme01 n'ont pas le même état <br>\n");
        }

?>

</body>
</html>
```

Toutes les aides présentes dans Java, C#, PHP 5 et Python, disparaissent du C++ qui, à nouveau, non seulement vous rend la vie plus compliquée, mais, pire encore, vous juge suffisamment aptes à affronter ces complications. Le test d’égalité des états d’objet et la possibilité de dupliquer des objets sont également présents dans C++. Cependant, ces différentes opérations sont à ce point attachées au mode de stockage des objets qu’il sera nécessaire, dans les codes et les explications qui suivent, d’étudier ce qui se passe en mémoire pile comme en mémoire tas. Le code qui suit s’en trouve considérablement allongé, et demande que l’on redouble d’attention par rapport à la version Java, Python ou PHP 5.

Traitement en surface et en profondeur

Les objets se référant mutuellement en mémoire, et constituant ainsi un graphe connecté dans cette même mémoire, il sera important, dans tout traitement qu’ils subissent, de penser à prolonger ces traitements le long du graphe ou pas, différenciant ainsi un traitement en surface d’un traitement en profondeur.

Égalité, clonage et affectation d’objets en C++

Code C++ illustrant la duplication, la comparaison et l’affectation d’objets

```
class O2 {
  private:
    int unAttribut02;
  public:
    O2(int unAttribut02) {
      this->unAttribut02 = unAttribut02;
    }
    int getAttribut() {
      return unAttribut02;
    }
    /* constructeur par copie */
```

```
/*
    O2(const O2& unO2) {
      unAttributO2 = unO2.unAttributO2;
   }
*/
    O2& operator=(const O2& unO2) { /* surcharge de l'affectation */
      unAttributO2 = unO2.unAttributO2;
      return *this;
    }
    /* déclaration de la surcharge de la comparaison */
/*
    friend bool operator==(const O2& unO2, const O2& unAutreO2);
*/
};
class O1 {
  private:
    int unAttributO1;
    O2* lienO2;
  public:
    O1(int unAttributO1, O2* lienO2){
      this->unAttributO1= unAttributO1;
      this->lienO2      = lienO2;
    }
    void donneAttribut(){
      cout <<"valeur attribut = " << unAttributO1 << endl;
    }
    /* constructeur par copie */
/*  O1(const O1& unO1) {
      unAttributO1      = unO1.unAttributO1;
      lienO2            = new O2(*unO1.lienO2);
    }
*/
    O1& operator=(const O1& unO1) { /* surcharge de l'affectation */
      unAttributO1 = unO1.unAttributO1;
      if (lienO2)
        delete lienO2;
      lienO2 = new O2(*unO1.lienO2);
      return *this;
    }
    /* déclaration de la surcharge de l'opérateur de comparaison */
/*
    friend bool operator==(const O1& unO1, const O1& unAutreO1);
*/
};
  /* redéfinition des opérations de comparaison */
/*
  bool operator==(const O2& unO2, const O2& unAutreO2) {
    if (unO2.unAttributO2 == unAutreO2.unAttributO2)
      return true;
    else return false;
  }
```

```
  bool operator==(const O1& unO1, const O1& unAutreO1) {
    if ((unO1.unAttributO1 == unAutreO1.unAttributO1)
        &&
        (unO1.lienO2->getAttribut()== unAutreO1.lienO2->getAttribut())
    )
      return true;
    else
      return false;
  }
*/

int main(int argc, char* argv[]) {
  /* Test de la méthode equal() */
  /* Objets créés dans le tas */
  O2* unO2Tas           = new O2(5);
  O2* unAutreO2Tas      = new O2(5);
  O2* unTroisiemeO2Tas  = new O2(10);
  O2* unQuatriemeO2Tas  = new O2(10);
  *unTroisiemeO2Tas     = *unO2Tas;
  unQuatriemeO2Tas      = unO2Tas;
  O2* unCinquiemeO2Tas  = new O2(*unO2Tas);

  O1* unO1Tas           = new O1(10, unO2Tas);
  O1* unAutreO1Tas      = new O1(10, unAutreO2Tas);
  O1* unTroisiemeO1Tas  = new O1(*unAutreO1Tas);
  O1* unQuatriemeO1Tas  = new O1(10,unO2Tas);
  *unQuatriemeO1Tas     = *unAutreO1Tas;

  if (unO2Tas           == unAutreO2Tas)
    cout << "unO2Tas et unAutreO2Tas ont la même référence" << endl;
  if (unO2Tas           == unTroisiemeO2Tas)
    cout << "unO2Tas et unTroisiemeO2Tas ont la meme reference" << endl;
  if (unO2Tas           == unQuatriemeO2Tas)
    cout << "unO2Tas et unQuatriemeO2Tas ont la meme reference" << endl;
  if (unO2Tas           == unCinquiemeO2Tas)
    cout << "unO2Tas et unCinquiemeO2Tas ont la meme reference" << endl;
  if (unO1Tas           == unAutreO1Tas)
    cout << "unO1Tas et unAutreO1Tas ont la même référence" << endl;
  if (unAutreO1Tas      == unTroisiemeO1Tas)
    cout << "unO1AutreTas et unTroisièmeO1Tas ont la même référence" << endl;
  if (unAutreO1Tas      == unQuatriemeO1Tas)
    cout << "unO1AutreTas et unQuatriemeO1Tas ont la même référence" << endl;
  if (*unO2Tas          == *unAutreO2Tas)
    cout << "unO2Tas et unAutreO2Tas ont le meme etat" << endl;
  if (*unO2Tas          == *unTroisiemeO2Tas)
    cout << "unO2Tas et unTroisiemeO2Tas ont le meme etat" << endl;
  if (*unO2Tas          == *unQuatriemeO2Tas)
    cout << "unO2Tas et unQuatriemeO2Tas ont le meme etat" << endl;
  if (*unO2Tas          == *unCinquiemeO2Tas)
    cout << "unO2Tas et unCinquiemeO2Tas ont le meme etat" << endl;
```

```
  if (*un01Tas          == *unAutre01Tas)
    cout << "un01Tas et unAutre01Tas ont le meme etat" << endl;
  if (*unAutre01Tas     == *unTroisieme01Tas)
    cout << "unAutre01Tas et unTroisieme01Tas ont le meme etat" << endl;
  if (*unAutre01Tas     == *unQuatrieme01Tas)
    cout << "unAutre01Tas et unQuatrieme01Tas ont le meme etat" << endl;

  /* Objets créés dans la pile */
  O2 unO2Pile           = O2(5);
  O2 unAutreO2Pile      = O2(5);
  O2 unTroisiemeO2Pile  = O2(10);
  O2 &unQuatriemeO2Pile = unO2Pile;
  unTroisiemeO2Pile     = unO2Pile;
  O2 unCinquiemeO2Pile  = O2(unO2Pile);

  O1 unO1Pile           = O1(10, &unO2Pile);
  O1 unAutreO1Pile      = O1(10, &unAutreO2Pile);
  O1 unTroisiemeO1Pile  = O1(unAutreO1Pile);
  O1 unQuatriemeO1Pile  = O1(10, &unO2Pile);
  unQuatriemeO1Pile     = unAutreO1Pile;

  if (&unO2Pile         == &unAutreO2Pile)
    cout << "unO2Pile et unAutreO2Pile ont la meme reference" << endl;
  if (&unO2Pile         == &unTroisiemeO2Pile)
    cout << "unO2Pile et unTroisiemeO2Pile ont la meme reference" << endl;
  if (&unO2Pile         == &unQuatriemeO2Pile)
    cout << "unO2Pile et unQuatriemeO2Pile ont la meme reference" << endl;
  if (&unO2Pile         == &unCinquiemeO2Pile)
    cout << "unO2Pile et unCinquiemeO2Pile ont la meme reference" << endl;
  if (&unO1Pile         == &unAutreO1Pile)
    cout << "unO1Pile et unAutreO1Pile ont la meme reference" << endl;
  if (&unAutreO1Pile    == &unTroisiemeO1Pile)
    cout << "unO1AutrePile et unTroisiemeO1Pile ont la meme reference" << endl;
  if (&unAutreO1Pile    == &unQuatriemeO1Pile)
    cout << "unO1AutrePile et unQuatriemeO1Pile ont la meme reference" << endl;
  if (unO2Pile          == unAutreO2Pile)
    cout << "unO2Pile et unAutreO2Pile ont le meme etat" << endl;
  if (unO2Pile          == unTroisiemeO2Pile)
    cout << "unO2Pile et unTroisiemeO2Pile ont le meme etat" << endl;
  if (unO2Pile          == unQuatriemeO2Pile)
    cout << "unO2Pile et unQuatriemeO2Pile ont le meme etat" << endl;
  if (unO2Pile          == unCinquiemeO2Pile)
    cout << "unO2Pile et unCinquiemeO2Pile ont le meme etat" << endl;
  if (unO1Pile          == unAutreO1Pile)
    cout << "unO1Pile et unAutreO1Pile ont le meme etat" << endl;
  if (unAutreO1Pile     == unTroisiemeO1Pile)
    cout << "unAutreO1Pile et unTroisiemeO1Pile ont le meme etat" << endl;
  if (unAutreO1Pile     == unQuatriemeO1Pile)
    cout << "unAutreO1Pile et unQuatriemeO1Pile ont le meme etat" << endl;
  return 0;
}
```

Nous allons décrire ce code ligne par ligne. Initialement, de nombreuses instructions seront maintenues en commentaire, que nous ré-activerons au fur et à mesure de leur justification. D’abord, la classe `O2` a, comme à l’habitude, son attribut, son constructeur et sa méthode d’accès. Les trois méthodes qui suivent sont désactivées pour l’instant. Il s’agit du constructeur par copie, qui participe à la duplication des objets et joue un rôle équivalent à la méthode `clone()` de Java. Ensuite, nous trouvons la surcharge de l’opérateur d’affectation, qui permettra de réaliser l’assignation d’un objet dans un autre (ce qui requerra également une duplication de l’objet assigné et une destruction en partie de l’objet affecté).

Finalement, nous trouvons la surcharge de l’opérateur de comparaison `==`, de manière à lui permettre de comparer des objets entre eux (rôle équivalent à la méthode `equals()` en Java). Une différence importante de C++ par rapport à Java est la mise en œuvre du mécanisme de surcharge d’opérateur (mécanisme qui existe également en C# et est très proche de son mode d’emploi en C++ ; nous verrons à la fin du chapitre comment C# joint la pratique de Java à celle de C++) pour réaliser l’égalité et l’assignation d’un objet dans un autre.

> **Surcharge d’opérateur**
>
> La surcharge d’opérateur consiste en général à simplement étendre la portée des opérateurs unaires (tel « ++ ») ou binaires (telle l’addition) à de nouveaux types, par exemple, de nouvelles classes. Ces opérateurs sont généralement prédéfinis pour des types primitifs. On peut, par défaut, additionner et comparer des entiers ou des réels entre eux, mais C++ vous offre la possibilité de comparer et d’additionner des fleurs, des animaux, des proies, des footballeurs, pour autant que vous surchargiez les opérateurs correspondants pour ces nouvelles classes. Ainsi, vous pourriez définir l’addition de deux objets fleurs comme l’obtention d’une nouvelle fleur possédant un nombre de pétales égal à l’addition des deux objets fleurs, ou celle de deux footballeurs, comme l’obtention d’un troisième, aussi absurde cela soit-il, ayant comme numéro ou comme QI la somme des deux autres. En fait, la surcharge d’opérateur est un jeu d’écriture, sous la responsabilité du compilateur, qui traduira dans une forme d’utilisation classique l’emploi d’un de ces opérateurs dans un contexte nouveau. Nous verrons le comment et le pourquoi de la surcharge des opérateurs de comparaison (==) et d’affectation (=) dans la suite.

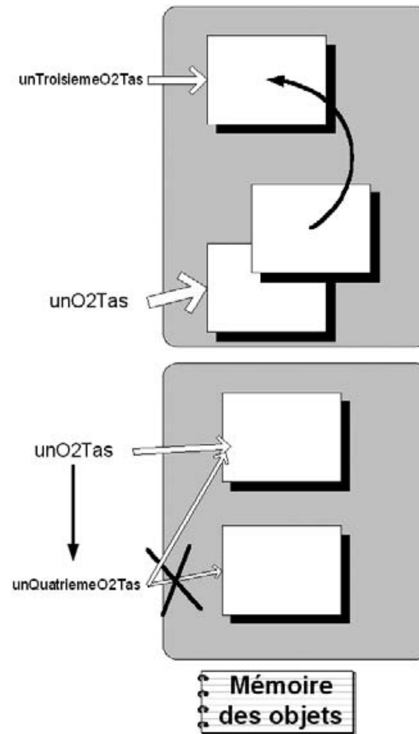
Traitons d’abord la mémoire tas

La classe `O1` ressemble à la classe `O2`, si ce n’est l’addition d’un référent vers un objet `O2`. De nouveau, on retrouve le constructeur par copie et la surcharge des opérateurs d’affectation et de comparaison. Passons maintenant à la partie principale, l’intérieur de la fonction `main()`. Tout d’abord, neuf objets sont créés dans la mémoire tas, cinq objets de la classe `O2` et quatre objets de la classe `O1`. Les quatre premiers objets `O2` sont créés sans surprise. Dans l’instruction suivante, `*unTroisiemeO2Tas = *unO2Tas`, le premier objet, dé-référé, c’est-à-dire l’objet vraiment, pas son adresse, est assigné au troisième, comme la figure suivante l’illustre.

Il y a, en conséquence, un clonage du premier objet qui s’opère, de manière à l’installer dans la mémoire, préalablement affectée au troisième. L’ancien objet, `unTroisiemeO2Tas`, a complètement disparu pour reproduire l’objet `unO2Tas`. Rien de bien particulier à signaler, car l’assignation d’un objet dans un autre se fait comme pour n’importe quelle variable dans n’importe quel langage de programmation (figure 14-4).

Dans l’instruction suivante, `unQuatriemeO2Tas = unO2Tas`, une autre affectation s’opère mais, cette fois- ci, ce sont les référents qui sont concernés et non plus les objets à proprement parler. Dorénavant, deux référents pointeront vers le même objet : `unQuatrièmeO2Tas` et `unO2Tas`. La figure 14-4 illustre la différence que présentent ces deux types d’assignation. Le dernier type est de façon classique celui que l’on rencontre pour les objets en Java et en C#. Dans ce cas, un objet, en l’absence de son référent, est en perdition en mémoire, et ne demande qu’à être récupéré par un « ramasse-miettes », malheureusement inexistant en C+ +.

Figure 14-4
Assignation de l'objet `un02Tas` *dans l'objet* `unTroisieme02Tas`*, et du référent de l'objet* `un02Tas` *dans le référent de l'objet* `unQuatrieme02Tas`*.*



L'instruction suivante, `02* unCinquieme02Tas =  new 02(*un02Tas)`, crée un nouvel objet, en utilisant le constructeur par copie de la classe `02`. Le constructeur par copie sert donc, à partir d'un référent vers un premier objet, ici `un02Tas`, à définir ce qu'il faut récupérer dans le premier pour le transmettre au second, ici `unCinquieme02Tas`. Il est obligatoire que ce constructeur par copie reçoive un référent comme argument car, s'il recevait un objet, il faudrait copier ce même objet, ce qui provoquerait un nouvel appel au constructeur par copie, et ainsi de suite *ad vitam aeternam* (même en latin, on préfère éviter cela en informatique). Le constructeur par copie, dont le rôle et le fonctionnement ne vont pas sans rappeler l'opérateur d'assignation, intervient principalement lors du passage d'un objet par argument dans une méthode quelconque, puisque à chaque appel de la méthode concernée un objet sera dupliqué.

Les deux instructions suivantes créent deux objets `01` en mémoire tas, en leur passant comme argument les référents vers deux objets `02`. L'instruction suivante crée `unTroisième01` en appelant à nouveau le constructeur par copie, mais sur le référent de l'objet `unAutre01Tas` cette fois. Finalement, un quatrième objet `01` est créé, mais auquel est assigné `unAutre01Tas` à la place. Les pointeurs étant dé-référés, ce sont de nouveaux les objets et non les référents qui sont concernés par cette assignation.

Surcharge de l'opérateur d'affectation

```
01& operator=(const 01& un01) { // surcharge de l'affectation
  unAttribut01 = un01.unAttribut01;
  if (lien02)
    delete lien02;
```

```
    lienO2 = new O2(*unO1.lienO2);
    return *this;
  }
```

Le petit code qui précède reprend la surcharge de l'opérateur d'affectation ou d'assignation, lorsqu'il porte sur les objets eux-mêmes, et non leur référent . Pour la classe O1, il s'agit tout d'abord de récupérer la valeur de l'attribut unAttributO1. Ensuite, il faut dupliquer l'objet O2 pointé vers l'attribut référent et l'installer dans l'objet assigné. Une étape importante est l'effacement de l'ancien objet référé par l'attribut référent. Étant donné que ce référent se mettra à pointer sur le nouvel objet O2, et en l'absence de tout ramasse-miettes, il est important d'effacer l'ancien objet O2 qui n'est plus référé par personne.

Comparaisons d'objets

Après la création de ces neuf objets, le code se lance dans des comparaisons de ces objets deux à deux. Les 7 premières comparaisons se font sur les référents. Cela ne pose aucun problème pour le compilateur car, comme les référents sont des adresses, il n'y a rien de gênant à les comparer deux à deux. Ces comparaisons se révéleront vraies à chaque fois que les deux référents pointeront vers un même objet. En revanche, les sept comparaisons qui suivent seront refusées par le compilateur (c'est pour cette raison que nous les plaçons en commentaire dans un premier temps). En effet, il s'agit maintenant de comparer, non plus des référents, mais des objets, et l'opérateur de comparaison n'est pas initialement prévu pour cela. On comprend, dès lors, que la seule façon d'éviter le joug du compilateur est de surcharger l'opérateur de comparaison pour les objets issus des classes O2 et O1.

La mémoire pile

Les neuf objets suivants sont créés dans la mémoire pile, mais de manière semblable à la création des objets tas. On retrouve de même les opérations de comparaison, d'abord entre les référents, ce que le compilateur accepte, ensuite entre les objets eux-mêmes, ce que le compilateur refuse à nouveau, sans surcharge de l'opérateur de comparaison.

À ce stade, le résultat de l'exécution est le suivant

```
unO2Tas et unQuatriemeO2Tas ont la même référence
unO2Pile et unQuatriemeO2Pile ont la même référence
```

En effet, le premier objet O2 et le quatrième partagent le même référent. Toutes les autres comparaisons sont, soit fausses, soit inactives, en l'absence de surcharge de l'opérateur de comparaison. Nous allons de ce pas le surcharger…

Surcharge de l'opérateur de comparaison

Comme indiqué dans le code, tant pour la classe O1 que la classe O2, l'opération se fait en deux temps. D'abord, a lieu la surcharge de l'opérateur en dehors des deux classes. La syntaxe de la signature, quelque peu alambiquée, est telle à faciliter par la suite le travail de récriture du compilateur :

```
  bool operator==(const O2& unO2, const O2& unAutreO2){
    if (unO2.unAttributO2 == unAutreO2.unAttributO2)
      return true;
```

```
    else
      return false;
  }
  bool operator==(const O1& unO1, const O1& unAutreO1){
    if (   (unO1.unAttributO1 == unAutreO1.unAttributO1)
        && (unO1.lienO2 != unAutreO1.lienO2)
        && (unO1.lienO2->getAttribut() == unAutreO1.lienO2->getAttribut())
    )
      return true;
    else
      return false;
  }
```

Comme cette opération de surcharge doit avoir accès aux attributs des classes `O1` et `O2`, déclarés `private`, une manière élégante d’autoriser cet accès est de déclarer la procédure de surcharge `friend` de la classe. C’est la raison de la présence de la signature de la surcharge, précédée du mot-clé `friend`, dans les deux classes.

Au même titre que les classes, des fonctions définies en dehors de toute classe (ce que C++ est le seul parmi les trois langages à permettre), peuvent être déclarées comme `friend`, si elles désirent un accès privilégié aux caractéristiques privées de ces classes. Vous pourrez constater que la comparaison, dans le cas de la classe `O1`, porte, non seulement sur son attribut, mais également sur l’attribut de l’objet référé. Les deux objets auront le même état si, bien qu’ils pointent vers des objets `O2` différents, toutes les valeurs d’attribut sont égales. Voyons le résultat, en réalisant la surcharge de l’opérateur de comparaison, autorisant, de ce fait, les comparaisons d’objet, que le compilateur nous laisse maintenant passer sans coup férir.

Résultat du code en rendant les comparaisons d’objet possible

```
unO2Tas et unQuatriemeO2Tas ont la même référence
unO2Tas et unAutreO2Tas ont le même état
unO2Tas et unTroisiemeO2Tas ont le même état
unO2Tas et unQuatriemeO2Tas ont le même état
unO2Tas et unCinquiemeO2Tas ont le même état
unO1Tas et unAutreO1Tas ont le même état
unO2Pile et unQuatriemeO2Pile ont la même référence
unO2Pile et unAutreO2Pile ont le même état
unO2Pile et unTroisiemeO2Pile ont le même état
unO2Pile et unQuatriemeO2Pile ont le même état
unO2Pile et unCinquiemeO2Pile ont le même état
unO1Pile et unAutreO1Pile ont le même état
```

Que constatons-nous ? Quel que soit leur mode de création, par assignation ou par constructeur par copie, les objets `O2` partagent le même état. Cela prouve qu’il existe, par défaut dans toute classe en C++, un constructeur par copie, qui se borne à recopier tous les attributs d’un objet à l’autre, et une opération d’affectation par défaut qui fait de même. Nous constatons, également, que les grands absents de ces comparaisons sont les objets `unAutreO1` et `unTroisiemeO1`, et `unAutreO1` et `unQuatriemeO1`.

À l’instar de ce qui se passait en Java pour la méthode `clone()`, cette comparaison et cette affectation par défaut ne permettent qu’un traitement en surface des objets. Les valeurs d’attributs sont dupliquées mais, dès le moment où un de ces attributs réfère une autre classe, il est important de se préoccuper de la duplication ou non des objets référés. D’autant qu’en C++, cela pourra vous éviter de mauvaise surprise, si vous effacez le référent dans un des objets, en oubliant qu’il est encore et toujours référé par l’objet affecté ou l’objet affectant. Ce n’est pas possible en Java, vu la présence heureuse du « ramasse-miettes », mais cela peut faire

énormément de dégâts en C++, dégâts dont la probabilité croît avec la taille du logiciel et la difficulté qu'il y a à suivre à la trace de multiples référents.

Dernière étape

Pour conclure avec le C++, nous allons enlever les derniers commentaires du code, ce qui revient à surcharger le constructeur par copie et l'opérateur d'affectation. Ces opérations de surcharge sont tellement répandues, sinon automatisées, que dans de nombreux générateurs de code automatique, à partir d'UML par exemple, elles sont ajoutées systématiquement dans le squelette de code C++ produit. On parle alors de la définition d'une « classe canonique ». Ci-après apparaît le code C++ de la classe `O1`, tel qu'il est automatiquement généré par le logiciel UML Rational Rose lors de la simple création d'une classe O1 dans le diagramme de classe. On y trouve un squelette de constructeur, de destructeur, mais tout est aussi en place pour pourvoir à la surcharge des opérateurs d'assignation et de comparaison (les deux comparaisons s'y trouvent, « == » et « != » qui signifie « non égal »). Pour Java, seules 2 ou 3 lignes sont générées – de quoi vous convaincre s'il était nécessaire encore, de la maîtrise accrue qu'exige la complexité du C++ par rapport à Java.

Code C++ de la classe O1 généré automatiquement par Rational Rose

```
//## begin module.cm preserve=no
//  %X% %Q% %Z% %W%
//## end module.cm
//## begin module.cp preserve=no
//## end module.cp
//## Module: O1; Pseudo Package body
//## Subsystem: <Top Level>
//## Source file: C:\Program Files\Rational\Rational Rose C++ Demo 4.0\O1.cpp
//## begin module.additionalIncludes preserve=no
//## end module.additionalIncludes
//## begin module.includes preserve=yes
//## end module.includes
// O1
#include "O1.h"
//## begin module.additionalDeclarations preserve=yes
//## end module.additionalDeclarations
// Class O1
O1::O1()
  //## begin O1::O1%.hasinit preserve=no
: unAttributO1(10)
  //## end O1::O1%.hasinit
  //## begin O1::O1%.initialization preserve=yes
  //## end O1::O1%.initialization
{
  //## begin O1::O1%.body preserve=yes
  //## end O1::O1%.body
}
O1::O1(const O1 &right)
  //## begin O1::O1%copy.hasinit preserve=no
: unAttributO1(10)
  //## end O1::O1%copy.hasinit
  //## begin O1::O1%copy.initialization preserve=yes
```

```
  //## end O1::O1%copy.initialization
{
  //## begin O1::O1%copy.body preserve=yes
  //## end O1::O1%copy.body
}
O1::~O1() {
  //## begin O1::~O1%.body preserve=yes
  //## end O1::~O1%.body
}
const O1 & O1::operator=(const O1 &right) {
  //## begin O1::operator=%.body preserve=yes
  //## end O1::operator=%.body
}
int O1::operator==(const O1 &right) const {
  //## begin O1::operator==%.body preserve=yes
  //## end O1::operator==%.body
}
int O1::operator!=(const O1 &right) const {
  //## begin O1::operator!=%.body preserve=yes
  //## end O1::operator!=%.body
}
//## Other Operations (implementation)
return O1::getAttribut(argtype argname) {
  //## begin O1::getAttribut%1021245027.body preserve=yes
  //## end O1::getAttribut%1021245027.body
}
// Additional Declarations
//## begin O1.declarations preserve=yes
//## end O1.declarations
```

Résultat final du code C++

```
unO2Tas et unQuatriemeO2Tas ont la même référence
unO2Tas et unAutreO2Tas ont le même état
unO2Tas et unTroisiemeO2Tas ont le même état
unO2Tas et unQuatriemeO2Tas ont le même état
unO2Tas et unCinquiemeO2Tas ont le même état
unO1Tas et unAutreO1Tas ont le même état
unAutreO1Tas et unTroisiemeO1Tas ont le même état
unAutreO1Tas et unQuatriemeO1Tas ont le même état
unO2Pile et unQuatriemeO2Pile ont la même référence
unO2Pile et unAutreO2Pile ont le même état
unO2Pile et unTroisiemeO2Pile ont le même état
unO2Pile et unQuatriemeO2Pile ont le même état
unO2Pile et unCinquiemeO2Pile ont le même état
unO1Pile et unAutreO1Pile ont le même état
unAutreO1Pile et unTroisiemeO1Pile ont le même état
unAutreO1Pile et unQuatriemeO1Pile ont le même état
```

Découvrons le résultat obtenu en supprimant tous les commentaires du code. On s'aperçoit que les égalités d'état se sont étendues à présent sur et entre tous les objets `O1`, qu'ils soient dans la pile ou dans le tas. Tant le clonage, l'affectation que les comparaisons se font maintenant en profondeur. Cette étude fouillée des méthodes

`equals()` et `clone()` de la superclasse `Object`, pour Java, de la version très simplifiée du Python, du rôle du constructeur par copie et des opérateurs de comparaison et d'affectation en C++ (C# permet au programmeur d'être mangé, comme nous allons le voir, indifféremment à la sauce Java ou à la sauce C++), avait un but principal.

Il s'agit d'enfoncer le clou sur la structure relationnelle des objets en mémoire, tant dans la pile que dans le tas. Toute opération de lecture, de sauvegarde (nous verrons cette sauvegarde au chapitre 19, mais, par avance, celle-ci, également, devra prendre grand soin à la structure relationnelle des objets entre eux), d'accès et d'effacement doit prendre en considération, et ce avec un maximum de soin et de prudence, tout le réseau relationnel des objets, au travers duquel voyage une ribambelle de messages. Un tel réseau est susceptible de s'installer extrêmement vite pendant l'exécution d'un programme, ce qui fait toute la puissance de l'OO, mais également sa fragilité, dès lors que le fonctionnement de ce réseau n'est pas suffisamment compris et maîtrisé.

En C#, un cocktail de Java et de C++

Comme le code présenté ci-après l'indique, la version C# permet d'opter indifféremment pour la manière Java, en redéfinissant la méthode `Equals()` et en définissant la méthode `Clone()`, ou pour la manière C++, en surchargeant les opérateurs appropriés. Nous avons vu que C# permet de stocker des objets dans la mémoire pile, en en faisant des instances de structure plutôt que de classe. Ces objets se créent comme ceux issus des classes, mais leur processus de destruction ainsi que les mécanismes d'affectation sont très différents. Dans le code, nous gardons les classes `O1` et `O2` pratiquement égales à celles définies dans Java, mais nous rajoutons les structures `SO1` et `SO2` pour traiter les objets créés dans la pile.

Pour les structures

Pour les structures, il n'y a pas lieu de s'occuper de surcharger quoi que ce soit en ce qui concerne l'assignation d'objets, puisque le fonctionnement par défaut est le seul que l'on puisse imaginer. L'assignation se fait d'office en suivant le fil des référents. Pour la comparaison, on peut soit surcharger les opérateurs de comparaison, soit redéfinir la méthode `Equals()` héritée de la superclasse `Object`. En effet, au même titre que les classes, les structures héritent également de la classe `Object`. C'est la seule classe dont elles peuvent hériter, les pauvres.

En général, C# vous incite à favoriser la redéfinition de la méthode `Equals()` par rapport à la surcharge d'opérateurs. Même si vous avez déjà surchargé les opérateurs appropriés, il vous avisera de redéfinir la méthode `Equals()` afin de pouvoir faire fonctionner la procédure de comparaison de manière polymorphique, vu que cette méthode provient de la classe `Object`. Vous ne pouvez, par ailleurs, pas surcharger l'opérateur `==` sans surcharger également son dual, l'opérateur `!=`. Intéressant, non ? C# vous oblige à ne pas mourir idiot et à rester cohérent : si vous surchargez un opérateur logique, vous êtes contraint et forcé de surcharger son contraire. Le compilateur en fait son affaire.

Pour les classes

Pour les classes, vous pouvez à nouveau utiliser la surcharge ou la redéfinition de la méthode `Equals()` pour la comparaison. En revanche, la duplication d'un objet ne peut se faire qu'à l'aide de la méthode `Clone()`, vu l'impossibilité de surcharger l'opérateur d'affectation ou d'assignation. Cette dernière procède d'abord à une copie superficielle de l'objet, par l'entremise de la méthode `MemberwiseClone()`, héritée de la superclasse `Object`, puis crée un clone afin de reproduire l'attribut référent. La présence `de MemberwiseClone()` se justifie par l'impossibilité en C# de redéfinir une méthode définie comme `protected` dans la superclasse, en assignant à la version redéfinie une priorité d'accés plus large (`public` ici).

Code C#

```
using System;

struct S01 {
  private int unAttribut01;
  private S02 lien02;

  public S01(int unAttribut01, S02 lien02) {
    this.unAttribut01 = unAttribut01;
    this.lien02 = lien02;
  }
  public S02 get02() {
    return lien02;
  }
  public void change02(int nouvelleValeur) {
    lien02.setAttribut(nouvelleValeur);
  }
  public void donneAttribut() {
    Console.WriteLine("valeur attribut = " + unAttribut01);
  }
  public static bool operator==(S01 un01, S01 unAutre01) { /* surcharge de l'opérateur de comparaison */
    if ((un01.unAttribut01 == unAutre01.unAttribut01)
        &&
        (un01.lien02.getAttribut() == unAutre01.lien02.getAttribut())
    )
      return true;
    else
      return false;
  }
  public static bool operator!=(S01 un01, S01 unAutre01) {
    if ((un01.unAttribut01 != unAutre01.unAttribut01)
        ||
        (un01.lien02.getAttribut() != unAutre01.lien02.getAttribut())
       )
      return true;
    else
      return false;
  }
  public override bool Equals(Object unObjet) { /* Redéfinition de la méthode Equals */
    if (unObjet != null) {
      if (unObjet is S01) {
        if ((unAttribut01 == ((S01)unObjet).unAttribut01)
            && (lien02.getAttribut() == ((S01)unObjet).lien02.getAttribut())
        )
          return true;
        else
          return false;
      }
    }
    return false;
  }
}
```

```
class O1 {
  private int unAttributO1;
  private O2 lienO2;

  public O1(int unAttributO1, O2 lienO2) {
    this.unAttributO1 = unAttributO1;
    this.lienO2 = lienO2;
  }
  public O2 getO2() {
    return lienO2;
  }
  public void donneAttribut() {
    Console.WriteLine("valeur attribut = " + unAttributO1);
  }
  public override bool Equals(Object unObjet) { /* redéfinition de la méthode Equals */
    if (this == unObjet) {
      return true;
    }
    else {
      O1 unAutreO1 = unObjet as O1;
      if (unObjet != null) {
        if (   (unAttributO1            == unAutreO1.unAttributO1)
            && (lienO2.getAttribut()    == unAutreO1.lienO2.getAttribut())
        )
          return true;
        else
          return false;
      }
    }
    return false;
  }
  public O1 Clone() { /* définition du clonage */
    O1 unNouveauO1 = (O1)this.MemberwiseClone();
    unNouveauO1.lienO2 = lienO2.Clone();
    return unNouveauO1;
  }
}
struct SO2 {
  private int unAttributO2;

  public SO2(int unAttributO2) {
    this.unAttributO2 = unAttributO2;
  }
  public int getAttribut() {
    return unAttributO2;
  }
  public void setAttribut(int nouvelleValeur) {
    unAttributO2 = nouvelleValeur;
  }
```

```
  public static bool operator==(S02 un02, S02 unAutre02) { /* surcharge de la comparaison*/
    if (un02.unAttribut02 == unAutre02.unAttribut02)
      return true;
    else
      return false;
  }
  public static bool operator!=(S02 un02, S02 unAutre02) {
    if (un02.unAttribut02 != unAutre02.unAttribut02)
      return true;
    else
      return false;
  }
  public override bool Equals(Object unObjet) { /* redéfinition de la méthode Equals */
    if (unObjet != null) {
      if (unObjet is S02) {
        if (unAttribut02 == ((S02)unObjet).unAttribut02)
          return true;
        else
          return false;
      }
    }
    return false;
  }
}
class 02 {
  private int unAttribut02;

  public 02(int unAttribut02) {
    this.unAttribut02 = unAttribut02;
  }
  public int getAttribut() {
    return unAttribut02;
  }
  public void setAttribut(int nouvelleValeur) {
    unAttribut02 = nouvelleValeur;
  }
  public override bool Equals(Object unObjet) { /* redéfinition de la méthode Equals */
    if (this == unObjet) {
      return true;
    }
    else {
      02 unAutre02 = unObjet as 02;
      if (unObjet != null) {
        if (unAttribut02 == unAutre02.unAttribut02)
          return true;
        else
          return false;
      }
    }
    return false;
  }
  public 02 Clone() {
    return((02) MemberwiseClone());
  }
}
```

```
public class CloneEqual {
  public static void Main() {
    /*   Test de la méthode equal()    */
    /*   Objets créés dans le tas */
    O2 unO2 = new O2(5);
    O2 unAutreO2 = new O2(5);
    O2 unTroisièmeO2 = new O2(10);
    unTroisièmeO2 = unO2;
    O1 unO1 = new O1(10, unO2);
    O1 unAutreO1 = new O1(10, unAutreO2);

    if (unO2 == unAutreO2)
      Console.WriteLine("unO2 et unAutreO2 ont la même référence");
    if (unO2.Equals(unAutreO2))
      Console.WriteLine("unO2 et unAutreO2 ont le même état");
    if (unO2 == unTroisièmeO2)
      Console.WriteLine("unO2 et unTroisièmeO2 ont le même état");
    if (unO2.Equals(unTroisièmeO2))
      Console.WriteLine("unO2 et unTroisièmeO2 ont le même état");
    if (unO1 == unAutreO1)
      Console.WriteLine("unO2 et unAutreO2 ont la même référence");
    if (unO1.Equals(unAutreO1))
      Console.WriteLine("unO1 et unAutreO1 ont le même état");

    /* Test de la méthode Clone  */
    O2 unQuatrièmeO2 = null;
    unQuatrièmeO2 = (O2)unTroisièmeO2.Clone();
    Console.WriteLine(unTroisièmeO2.getAttribut() + " = ? " + unQuatrièmeO2.getAttribut());

    O1 unTroisièmeO1 = null;
    unTroisièmeO1 = (O1)unAutreO1.Clone();

    if (unO1.Equals(unTroisièmeO1))
      Console.WriteLine("unO1 et unTroisièmeO1 ont le même état");
    unTroisièmeO1.getO2().setAttribut(7);
    if (unO1.Equals(unTroisièmeO1))
      Console.WriteLine("unO1 et unTroisièmeO1 ont le même état");
    else
      Console.WriteLine("unO1 et unTroisièmeO1 n'ont pas le même état");

    /* Objets créés dans la pile */
    SO2 unO2Pile = new SO2(5);
    SO2 unAutreO2Pile = new SO2(5);
    SO2 unTroisiemeO2Pile = new SO2(10);
    unTroisiemeO2Pile = unO2Pile;

    SO1 unO1Pile = new SO1(10, unO2Pile);
    SO1 unAutreO1Pile = new SO1(10, unAutreO2Pile);
    SO1 unTroisiemeO1Pile = unAutreO1Pile;

    if (unO2Pile == unAutreO2Pile)
      Console.WriteLine("unO2Pile et unAutreO2Pile ont le meme etat");
```

```
      if (un02Pile == unTroisieme02Pile)
        Console.WriteLine("un02Pile et unTroisieme02Pile ont le meme etat");
      if (un01Pile == unAutre01Pile)
        Console.WriteLine("un01Pile et unAutre01Pile ont le meme etat");
      if (un01Pile.Equals(unAutre01Pile))
        Console.WriteLine("un01Pile et unAutre01Pile ont le meme etat");
      if (unAutre01Pile == unTroisieme01Pile)
        Console.WriteLine("un01AutrePile et unTroisieme01Pile ont le meme   etat");
      unTroisieme01Pile.change02(7);
      if (un01Pile == unTroisieme01Pile)
        Console.WriteLine("un01Pile et unTroisième01Pile ont le même état");
      else
        Console.WriteLine("un01Pile et unTroisième01Pile n'ont pas le même état");
    }
  }
```

Résultat

```
un02 et unAutre02 ont le même état
un02 et unTroisième02 ont le même état
un02 et unTroisième02 ont le même état
un01 et unAutre01 ont le même état
5 = ? 5
un01 et unTroisième01 ont le même état
un01 et unTroisième01 n’ont pas le même état
un02Pile et unAutre02Pile ont le même état
un02Pile et unTroisieme02Pile ont le même état
un01Pile et unAutre01Pile ont le même état
un01Pile et unAutre01Pile ont le même état
un01AutrePile et unTroisieme01Pile ont le même état
un01Pile et unTroisième01Pile n’ont pas le même état.
```

Le code étant très proche des codes Java et C++ précédents, le résultat devrait apparaître très logique. On s’aperçoit que l’objet `unTroisieme01` est bien une copie en profondeur de l’objet `un01` car, en changeant la valeur de l’attribut pointé par le premier, l’égalité n’a plus court. Il en va de même pour les objets pile `un01Pile` et `unTroisième01Pile`, bien que nulle redéfinition de mécanisme d’affectation n’ait été nécessaire. La répétition de la phrase « `un01Pile` et un `Autre01Pile` ont le même état » est dû à ce que l’on a effectué la comparaison des deux manières possibles proposées par C#, soit à l’aide de l’opérateur == surchargé, soit à l’aide de la méthode `Equals()` redéfinie.

Exercices

Exercice 14.1

Créez une classe `CompteEnBanque` munie d'un seul attribut `solde`, et définissez l'égalité de deux objets comptes en banque, de telle manière qu'elle soit vérifiée dès lors que les deux soldes sont égaux. Réalisez cet exercice en C# et en C++.

Exercice 14.2

Surchargez dans ces deux mêmes langages l'opérateur d'addition pour cette même classe, de telle manière que la somme de deux comptes en banque en donne un troisième, dont le solde est la somme des deux soldes.

Exercice 14.3

Pourquoi Java a-t-il déclaré `protected` la méthode `equals()` dans la superclasse `Object` ?

Exercice 14.4

Pourquoi C# a-t-il déclaré `static` la version de la méthode `Equals(object a, object b)` dans la superclasse `Object`, et qui compare les deux objets reçus en tant qu'argument ?

Exercice 14.5

Pourquoi est-il plus dans l'esprit OO en C# de redéfinir la méthode `Equals()` plutôt que de surcharger l'opérateur `==` ?

Exercice 14.6

Créez une nouvelle classe `Emprunt` munie d'un seul attribut `montant` et modifiez la classe `CompteEnBanque` de telle manière que plusieurs objets emprunts puissent être associés à un même objet compte en banque. Redéfinissez l'égalité de deux comptes en banque comme vérifiée, si la somme des montants des emprunts est égale dans les deux cas.

Exercice 14.7

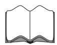
Redéfinissez la méthode `clone()` en Java pour la classe `CompteEnBanque` de manière que les emprunts se trouvent également clonés lors du clonage du compte.

15

Interfaces

Ce chapitre présente les interfaces, structures de code qui se bornent à introduire les seules signatures des méthodes. Il décrit dans les quatre langages de programmation qui en font usage les trois rôles que ces interfaces sont appelées à jouer : forcer l'implémentation de leurs méthodes, permettre le multihéritage, faciliter et stabiliser la décomposition de l'application logicielle.

Sommaire : Interface — Multihéritage d'interfaces en Java, C# et PHP 5 — Interface = contrat de médiation — L'essor des interfaces dans UML 2 — Fichiers .h et fichiers .cpp en C++ — Décomposition de l'application



Candidus — Ce mode d'emploi que représente l'interface, s'agit-il en quelque sorte d'une liste d'ingrédients permettant d'identifier un médicament générique ?

Doctus — Ton image est bonne. Tout ce qu'on attend d'un objet peut être exprimé par son interface. Seuls les attributs nécessaires à la concrétisation des objets en sont exclus.

Cand. — Les interfaces ne sont-elles qu'une façade publique pour l'implémentation ? N'ont-ils aucun impact sur l'exécution des programmes ?

Doc. — Bien sûr que si. Une interface joue certains des rôles d'une classe, elle représente un sous-ensemble de méthodes – que tu pourras invoquer comme s'il s'agissait d'une classe à part entière. Même le compilateur se contente des signatures qu'elles contiennent. Tu peux compiler une classe dépendant d'une interface avant même d'avoir réalisé la moindre classe d'implémentation concrète.

Cand. — S'il ne s'agit que de listes de leurs signatures, on peut imaginer que n'importe quel sous-ensemble des méthodes d'une classe suffit à définir une interface ?

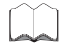
Doc. — Absolument. C'est même cette simplicité qui nous permet de réaliser en Java quelque chose d'équivalent au multihéritage. Un multihéritage réel repose sur une simple déclaration de parenté multiple donnée au compilateur. Java n'autorisant qu'une seule classe parent, tu devras recourir aux interfaces pour obtenir du compilateur qu'il te demande d'implémenter toutes les méthodes déclarées par une classe.

Cand. — Une interface ne serait donc qu'une classe dont toutes les méthodes sont abstraites ?

Doc. — Attention tout de même ! En Java, tu pourras implémenter plusieurs interfaces mais tu ne pourras hériter que d'une seule superclasse.

Cand. — Si je pousse à l'extrême, juste pour voir, il devrait donc être possible de créer un objet ne dépendant que d'un ensemble d'interfaces. Il ne dépendra alors jamais directement des classes concrètes avec lesquelles il doit communiquer.

Doc. — Rien ne l’interdit en effet, à tel point que de tels objets peuvent être mis en œuvre sur un réseau. Chacun des ordinateurs ne devra disposer que des seules interfaces nécessaires pour la communication. Pour envoyer une requête à un objet situé sur une machine distante, les seules signatures de méthodes de cet objet suffiront. Le message sera envoyé à l’objet concret de la machine distante qui se chargera d’exécuter la méthode associée.



Peter Coad et TogetherJ

Comme nous recourrons plus d’une fois à TogetherJ dans cet ouvrage, pour la réalisation des diagrammes UML, le moins que nous puissions faire est de donner un grand coup de chapeau à Peter Coad, le créateur de TogetherSoft, l’entreprise qui produit ce remarquable logiciel et depuis rachetée par Borland (dont Coad fut le vice-président jusqu’en 2004), qui s’est précipité de l’intégrer dans ses nouveaux environnements de développement. Ce qui nous impressionne le plus dans l’utilisation de ce logiciel est la synchronisation qu’il fut le premier à permettre entre le code (Java en l’occurrence, mais des versions du logiciel existent pour d’autres langages, y compris C#, C++ ou VB.Net) et les diagrammes UML 2. Changez quoi que ce soit dans un diagramme de classes et le code « encaisse » immédiatement ce changement, changez quoi que ce soit dans le code et les diagrammes UML s’adaptent pour intégrer cette modification, et tout cela immédiatement devant vos yeux ébahis. TogetherJ fut le premier à parvenir à cette complète synchronisation, suivi depuis par d’autres environnements de développement UML, à l’instar d’un de ses concurrents les plus sérieux comme Rational Rose, Umondo, Argo et beaucoup d’autres. Cette synchronisation est à ce point effective, qu’il nous est arrivé de perdre toute une partie de code car nous avions, sans y prendre garde, effacé quelques rectangles dans le diagramme de classe. De fait, cette synchronisation est discutée par pas mal de praticiens, selon le statut que l’on accorde aux diagrammes UML (nous approfondissons ce point dans le chapitre 10). Cela peut se limiter (mais c’est déjà indispensable) à une aide à la conception, et qui se maintient quelque peu à « l’écart » de la réalisation matérielle finale, un peu comme un plan d’architecte ou une partition de musique, qu’il est impensable d’introduire dans une quelconque machine afin que puisse en résulter la maison ou la symphonie. Mais la voie tracée par Peter Coad est d’en faire un vrai outil de développement qui prend une part active à l’obtention des produits finaux, comme une partition musicale que l’on enrichirait de tout ce qui est nécessaire à la production de la musique. Les deux visions coexistent, mais la seconde (que nous avons poussée à l’extrême dans le chapitre 10 en assimilant UML 2 à un nouveau type de langage de programmation) semble gagner chaque jour un peu plus de terrain. C’est clairement celle qui accompagne le MDA de l’OMG.

Peter Coad n’a pas réellement besoin de cette publicité tant celle-ci est omniprésente sur le Web, produite par d’autres ou, plus effectivement encore, par le principal intéressé. On retrouvait, par exemple, sur la page Web de TogetherSoft, des annonces rien de moins emphatiques telles que « Peter Coad is a business builder, a model builder and a thought leader », excusez du peu. Avec cette société, il est vrai, Peter Coad n’en est pas à son premier coup de maître. Il est reconnu comme un gourou OO (auteur de la méthodologie Coad/Yourdon pour l’analyse orientée objet) depuis pas mal d’années, et a produit un ensemble d’ouvrages dans la même maison d’édition, Prentice Hall, intitulés : *Java Modeling in Color with UML* – un livre où Peter Coad colorie les diagrammes UML pour y intégrer des informations temporelles sur le développement des parties de ces diagrammes –, et plusieurs autres comme *Java Design : Building Better Apps and Applets* ou *Object Models: Strategies, Patterns and Applications*, dans lesquels il développe sa vision de la modélisation objet, avec des points forts et discutables à la fois, comme la préférence qu’il accorde à la composition sur l’héritage (et que nous discutons plusieurs fois dans ce livre). Par ailleurs, il dirige une collection dans cette même maison d’édition.

Il dépense aussi une énergie folle à promouvoir un type de développement (qui, bien sûr, privilégie l’usage des logiciels Together), reposant sur un ensemble de principes devenus très populaires ces jours-ci et tenant de l’Extreme Programming[a], comme d’encourager les développeurs à produire très vite et à fréquence élevée des exécutables, modestes dans leur portée, mais de haute qualité ; mais aussi à intégrer plusieurs acteurs dans le processus de développement, le client et les experts du domaine, bien sûr les programmeurs, et à les faire communiquer au mieux (UML est là pour cela). Pour cela, l’analyse, le design, la programmation et la mise à l’essai doivent être faits de manière quasi simultanée (d’où l’utilisation de logiciel intégrateur comme Together). Surtout, le développement doit se dérouler en une succession de courtes itérations, débouchant à chaque fois sur un produit exécutable et facilement évaluable. Plus récemment, Peter Coad se fait connaître grâce au système d’enseignement et d’assistance qu’il offre à ses clients, lecteurs de la Bible, de sorte à pouvoir lire celle-ci dans sa langue d’origine. De la bible « OO » à celle « JC »… Ah, ces Américains !

a. *L’Extreme Programming*, Bénard *et al.*, Eyrolles 2002.

Interfaces : favoriser la décomposition et la stabilité

Nous avons entr'aperçu les interfaces, dans un chapitre précédent, comme une structure de code dont la finalité première est d'extraire d'une classe l'ensemble des signatures de ses services, afin d'en informer toutes celles qui voudraient y faire appel. Ces classes n'ont nul besoin des détails d'implémentation, c'est-à-dire de la manière précise dont ces services seront réellement exécutés (le corps d'instruction) par l'objet qui les fournit. Il suffira d'appeler le service par son nom pour le voir s'exécuter.

L'utilisation d'interfaces conduit naturellement à des applications facilement décomposables, réparties à travers une large équipe de programmeurs, tout en permettant une forte résistance aux changements d'implémentation. Ce sont les interfaces qui circuleront de programmeur en programmeur car ce sont les seuls éléments de code dont chacun d'eux, occupé à la réalisation de sa classe, a besoin dans son interaction avec les autres classes. Une fois sa classe bien entamée, ce programmeur en extraira également les services qu'il doit rendre disponibles aux autres. C'est l'aboutissement naturel de l'encapsulation, quand elle est pratiquée à l'extrême. On dissimule tout ce qui concerne l'implémentation des classes, au point d'en faire un fichier distinct et inaccessible, au contraire du fichier reprenant les noms des seuls services rendus par cette classe, qui est disponible, lui, pour tout utilisateur.

Java, C# et PHP 5 : interface via l'héritage

En Java, C# et PHP 5 (Python n'intègre pas dans sa syntaxe cette structure de données, tout comme il ne permet pas les méthodes abstraites), des classes abstraites aux interfaces il n'y a qu'un pas, puisqu'une interface est une classe abstraite dont toutes les méthodes sont déclarées abstraites. Nous verrons plus avant dans ce chapitre qu'en matière d'interface, C++ voit les choses un peu différemment. Tout en conservant les avantages qu'elles permettent dans la décomposition et la stabilisation des applications logicielles, C++ a détaché la pratique des interfaces de l'héritage.

En Java, les seuls attributs pouvant encore figurer dans la définition d'une interface sont `public`, `final` (c'est-à-dire constant, une fois leur valeur déterminée ils ne sont plus modifiables) et `static` (ce qui est assez logique, puisque, à l'instar d'une classe abstraite, vous ne pouvez créer des objets instances des interfaces). C# et PHP 5 n'autorisent aucun attribut dans la définition de ses interfaces. En Java et PHP 5, on n'hérite pas d'une interface, mais on l'implémente, en respectant la syntaxe suivante :

```
class O1 extends O2 implements IO2
```

désignant une classe `O1` qui hérite d'une classe `O2` et implémente une interface `IO2`. C# ne fait pas de différence syntaxique entre l'héritage de classe et d'interface, et la version C# de l'instruction précédente se réduit simplement à :

```
class O1 : O2 , IO2
```

à ceci près que les interfaces doivent être héritées en dernier, c'est-à-dire à la suite de la seule classe dont on peut hériter.

Les interfaces peuvent normalement hériter entre elles, comme le diagramme de classe présenté ci-après l'indique. En UML, le lien continu représente l'héritage et le lien en pointillé représente l'implémentation (C# ne faisant pas de différence entre les deux). Quand ils concernent les interfaces, les graphes d'héritage peuvent être plus complexes que quand ils se limitent aux seules classes. Ces graphes peuvent, en effet, présenter des ramifications multiples, tant descendantes qu'ascendantes (sinon ils se restreindraient à des structures d'arbre).

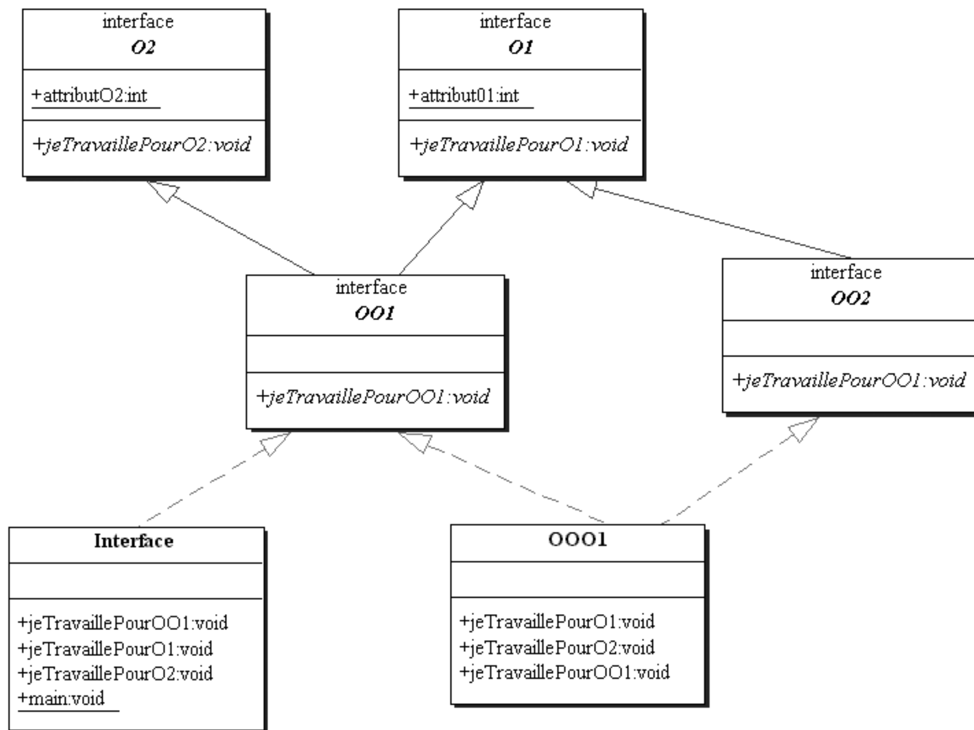


Figure 15-1

Structure d'héritage des interfaces. Dans ce diagramme de classes UML, figurent quatre interfaces O2, O1, OO1 et OO2, et deux classes, OOO1 et Interface.

Les trois raisons d'être des interfaces

Forcer la redéfinition

Les interfaces ont principalement trois raisons d'être. La première, largement exploitée par Java, est de forcer le programmeur à réutiliser des fonctionnalités déjà prédéfinies dans les librairies Java, librairies d'utilitaires écrites très souvent sous forme d'interfaces. Par exemple, toute classe Java peut implémenter l'interface `MouseListener`, dont toutes les méthodes abstraites seront déclenchées par des événements souris, comme `public void mouseClicked()` ou `public void mouseReleased()`.

En implémentant l'interface `MouseListener`, vous voilà contraints et forcés d'exprimer, par la concrétisation de ces méthodes abstraites, ce que votre code exécutera en réponse à un clic de souris. En gros, en cliquant sur la souris, vous passez la main au système d'exploitation de votre ordinateur, qui cherche par quelle voie redonner la main au programme. Il le fera en se servant des méthodes appropriées, telle `mouseClicked()` (qui s'exécute d'un simple clic de souris), que vous aurez concrétisées dans votre programme. D'autres interfaces très utiles, comme `Runnable` (pour la redéfinition de la méthode `public void run()` qui contient le corps d'instruction à exécuter par un thread [voir chapitre 17]) ou `KeyListener` (pour l'utilisation du clavier), sont souvent implémentées ensemble par une même classe (d'où la nécessité d'autoriser la multi-implémentation d'interfaces).

Implémentation d'interfaces

Quand, en Java, C# et PHP 5, une classe implémente ou hérite d'une interface, elle est contrainte et forcée de concrétiser les méthodes qu'elle hérite de celle-ci. Ne pas le faire génère une erreur lors de la compilation ou directement à l'exécution pour PHP 5. L'interface oblige à utiliser et à redéfinir ses méthodes. En Java, c'est par le biais des interfaces que sont implémentées les fonctionnalités GUI et multithread. Nous verrons que C# et Python procèdent différemment.

Dans la suite, nous présentons trois petits codes Java, ayant pour mission d'illustrer l'utilisation de trois interfaces : `Comparable`, `ActionListener` et `KeyListener`. Nous vous renvoyons aux manuels de programmation Java (ils ne manquent pas sur les étals des librairies ou sur le Web) pour approfondir la composition et l'utilisation de ces interfaces. Nous y faisons juste une rapide allusion pour comprendre comment ces structures d'interface peuvent jouer un rôle majeur dans le développement d'utilitaires et l'exploitation de bibliothèques Java existantes.

Le premier code illustre bien l'utilisation des interfaces afin de récupérer une fonctionnalité existante fort utile, le « triage de liste ». Il est souvent indispensable de trier toute liste par ordre croissant ou décroissant des éléments qui la composent. Or les algorithmes de tri font la joie des programmeurs tant ceux-ci peuvent être sujets à d'incroyables variations, allant de la plus extrême maladresse et naïveté du programmeur débutant à la plus subtile sophistication d'un hacker endiablé. La différence se mesurera par le temps de calcul nécessaire à la réalisation du tri par l'algorithme en question. C'est pourquoi Java vous épargne la tâche de réécrire la méthode de tri (il offre une excellente méthode de tri appelée `sort` dans l'interface `Collections`). Cependant, reste encore – ce que Java ne peut faire pour vous – à définir votre critère de tri par l'implémentation de l'interface `Comparable` et la redéfinition obligatoire de la méthode `compareTo` que cette interface possède. Cette méthode est censée renvoyée 1, -1 ou 0, selon le résultat de la comparaison. Dans le code qui suit, il s'agit de trier des objets de la classe O1 sur la valeur de leur seul attribut.

Code Java illustrant l'interface Comparable

```
import java.util.*;
class O1 implements Comparable {
   private int a;

   O1(int a) {
      this.a = a;
   }
   public void printAttribute(){
      System.out.print(a + " ");
   }
   public int compareTo(Object o) { // obligation de la redéfinir
      if (a > ((O1)o).a){
         return 1;}
      else
         if (a < ((O1)o).a){
            return -1;}
         else {
            return 0;
         }
   }
}
```

```
public class TestInterface {
   public static void main(String[] args){
      ArrayList lesO = new ArrayList();
      lesO.add(new O1(10));
      lesO.add(new O1(25));
      lesO.add(new O1(5));
      lesO.add(new O1(42));

      System.out.println("Liste non triée");
      for (int i=0; i<lesO.size(); i++){
         ((O1)lesO.get(i)).printAttribute();
      }

      Collections.sort(lesO); // le tri est effectué

      System.out.println();
      System.out.println("Liste triée");
      for (int i=0; i<lesO.size(); i++){
         ((O1)lesO.get(i)).printAttribute();
      }
   }
}
```

Résultat

```
Liste non triée
10 25 5 42
Liste triée
5 10 25 42
```

Le deuxième code concrétise une méthode abstraite provenant de l’interface « ActionListener » dont il hérite, qui se déclenche en cliquant sur un objet graphique, tel un bouton. Lorsque vous l’exécutez, une fenêtre apparaît avec trois boutons. En cliquant sur un des boutons, vous changez le « look » de votre application. Bien que plusieurs éléments de ce code soient très liés aux librairies graphiques Java, que nous ne voyons pas ici, le fonctionnement devrait rester compréhensible. L’unique mécanisme que nous cherchons à mettre en évidence, c’est l’obligation d’utiliser et de redéfinir une méthode (ici `public void actionPerformed()`) héritée de l’interface « ActionListener » où elle est abstraite, afin d’indiquer ce qui se passera lors d’un clic sur un des trois boutons. Le résultat du code apparaît figure 15-2.

Code Java illustrant l’interface ActionListener

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
class PlafPanel extends JPanel implements ActionListener { /* on implémente l'interface ActionListener */
  private JButton metalButton;
  private JButton motifButton;
  private JButton windowsButton;

  public PlafPanel() {
    /* on ajoute trois boutons */
    metalButton = new JButton("Metal");
    motifButton = new JButton("Motif");
    windowsButton = new JButton("Windows");
    add(metalButton);
    add(motifButton);
    add(windowsButton);
    /* on rend ces boutons sensibles à ce qui est dit
       dans les méthodes redéfinies à partir de l'interface, c'est-à-dire
        la méthode « actionPerformed » */
    metalButton.addActionListener(this);
    motifButton.addActionListener(this);
    windowsButton.addActionListener(this);
  }
  /* la méthode à absolument redéfinir, et qui dit ce qui se passe en cliquant sur les boutons */
  public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    String plaf = " ";

    if (source == metalButton) // Teste pour savoir de quel bouton il s'agit.
      plaf = "javax.swing.plaf.metal.MetalLookAndFeel";
    else if (source == motifButton)
      plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
    else if (source == windowsButton)
      plaf = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
    try {
      UIManager.setLookAndFeel(plaf);
      SwingUtilities.updateComponentTreeUI(this);
    }
    catch(Exception e) {}
  }
}
public class ExempleDInterface1 extends JFrame { /* JFrame est la fenêtre qui apparaît à l'exécution
             ➥du code */
  public ExempleDInterface1() {
    setTitle("Test Plateforme"); // titre de la fenêtre
    setSize(300, 200); // taille de la fenêtre

    addWindowListener(
      new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            System.exit(0);
```

```
          }
        }
      );
      getContentPane().add(new PlafPanel()); /* on ajoute sur le JFrame le panneau avec les trois
            ➥boutons */
    }
    public static void main(String[] args) {
      ExempleDInterface1 unTest = new ExempleDInterface1();
      unTest.setVisible(true); // on fait apparaître la fenêtre
    }
  }
```

Figure 15-2
Illustration de l'interface ActionListener. Quand on clique sur un des trois boutons, l'apparence de l'application change.

Résultat

Le troisième code, illustrant le rôle et l'utilisation de l'interface `KeyListener`, vous permet de vous servir des quatre touches fléchées du clavier, afin de dessiner à l'écran en une succession de traits verticaux et horizontaux. Si vous appuyez sur la touche « shift » en même temps, le dessin sera exécuté plus rapidement. Là encore, la connaissance des librairies graphiques Java est indispensable à une compréhension complète du code (et sort largement du cadre de cet ouvrage, un petit manuel Java vous sera utile), mais les fonctionnalités principales, et surtout la redéfinition des méthodes liées au clavier et à l'interface `KeyListener`, devraient apparaître suffisamment compréhensibles. Trois méthodes abstraites sont définies dans cette interface. L'implémentation de l'interface oblige la redéfinition des trois méthodes, y compris si nous n'avons rien de précis à demander à certaines d'entre elles (comme c'est le cas dans ce code). Le résultat du code est illustré figure 15-3.

Code Java illustrant l'interface KeyListener

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class SketchPanel extends JPanel implements KeyListener { // l'interface qui nous importe
  private Point start,end;

  public SketchPanel() {
    start = new Point(0,0);
    end = new Point(0,0);
    addKeyListener(this);
  }
```

```
  /* une méthode à nécessairement redéfinir, mais la seule que l'on redéfinit vraiment */
  public void keyPressed(KeyEvent evt) {
    int keyCode = evt.getKeyCode();
    int modifiers = evt.getModifiers();
    int d;

    if ((modifiers & InputEvent.SHIFT_MASK) != 0) // si on appuie sur Shift
      d = 5;
    else
      d = 1;
    if (keyCode == KeyEvent.VK_LEFT) add (-d,0); // selon la touche sur laquelle on appuie
    else if (keyCode == KeyEvent.VK_RIGHT) add (d,0);
    else if (keyCode == KeyEvent.VK_UP) add (0,-d);
    else if (keyCode == KeyEvent.VK_DOWN) add (0,d);
  }
  public void keyReleased(KeyEvent evt) {}; /* on la redéfinit sans vraiment
                                              la redéfinir, sinon gare au compilateur */

  public void keyTyped(KeyEvent evt) {}; /* on la redéfinit sans vraiment la
                                           redéfinir */

  public boolean isFocusable() { return true; }

  public void add (int dx, int dy) { // c'est la méthode qui trace les lignes
    end.x += dx;
    end.y += dy;
    Graphics g = getGraphics();
    g.drawLine(start.x, start.y, end.x, end.y);
    g.dispose();
    start.x = end.x;
    start.y = end.y;
  }
}
public class ExempleDInterface2 extends JFrame {
  public ExempleDInterface2() {
    setTitle("Test Jeu Graphique");
    setSize(300, 200);
    addWindowListener(
        new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            System.exit(0);
          }
        }
    );
    getContentPane().add(new SketchPanel());
  }
```

```
  public static void main(String[] args) {
    ExempleDInterface2 unTest = new ExempleDInterface2();
    unTest.setVisible(true);
  }
}
```

Résultat

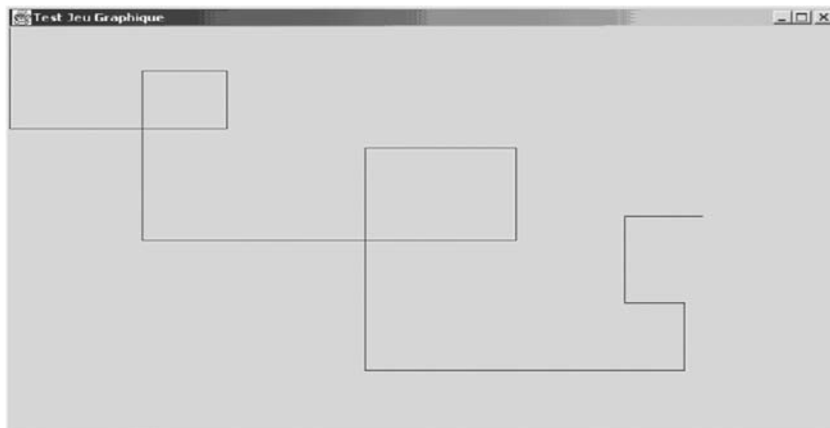


Figure 15-3
Illustration de l’interface KeyListener. En utilisant les 4 touches fléchées, vous pouvez dessiner au moyen d’une succession de traits horizontaux et verticaux.

Permettre le multihéritage

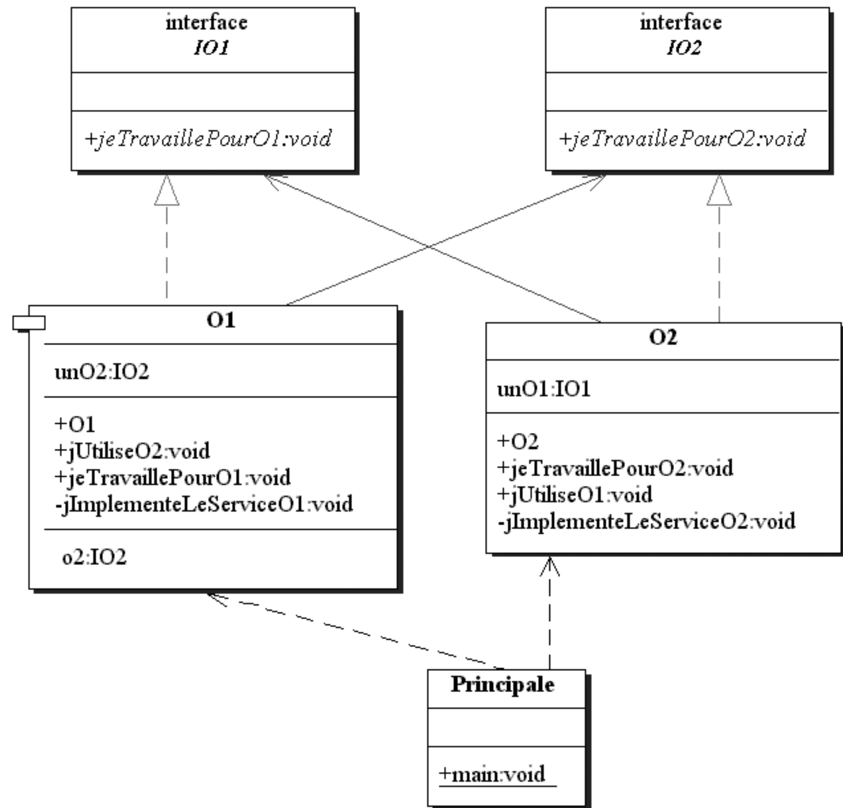
Comme nous l’avons dit, plusieurs interfaces seront très souvent implémentées en même temps. De fait, un autre apport des interfaces est d’avoir levé l’interdiction du multihéritage : d’interdit pour les classes, il devient autorisé pour les interfaces. Il s’agit plutôt d’une multi-implémentation en Java et PHP 5, quand une classe implémente plusieurs interfaces, mais c’est vraiment un multihéritage, quand une interface en hérite de plusieurs autres ou, dans tous les cas de figure, en C#.

La disparition du corps d’instruction dans les méthodes permet de contourner toutes les difficultés posées par le multihéritage en C++ (discutées au chapitre 11). Comme dans le diagramme UML précédent, rien n’interdit deux interfaces de posséder deux signatures de méthodes égales (dans ce diagramme, la même signature de méthode `jeTravaillePour001()` apparaît deux fois), puisque seule la classe, plus bas dans le graphe d’héritage, fournira un contenu à ces méthodes. De même, une classe et une interface pourraient partager la même signature de méthode. Si elles se trouvent héritées par une sous-classe commune, c’est de la seule version concrétisée de la méthode qu’héritera réellement la sous-classe. Concernant les attributs, leur déclaration restreinte à des constantes statiques en Java, et leur disparition pure et simple du C# et PHP 5, réduisent également les problèmes posés par des noms égaux.

La carte de visite de l’objet

Finalement, l’interface peut s’assimiler à la carte de visite d’une classe (et de tous les objets auxquels elle donne naissance), qu’un développeur d’une autre classe consulte et s’engage à respecter dans la conception de la sienne. Le diagramme UML qui suit illustre une collaboration idéale entre deux programmeurs, le premier de la classe `O1` et l’autre de la classe `O2`.

Figure 15-4
Diagramme de classes d'interaction entre les classes O1 et O2, passant par la médiation des deux interfaces implémentées par les classes IO1 et IO2.



Ce qui apparaît dans ce diagramme, c'est que l'unique dépendance de la classe `O1` avec la classe `O2` passe par un lien avec la seule interface de la classe `O2`, c'est-à-dire `IO2`. L'unique fichier dont doit disposer le développeur de la classe `O1` est un fichier contenant le code de l'interface `IO2`, et aucun autre. L'existence d'une implémentation quelconque de cette interface par une classe, ici la classe `O2`, peut ne le préoccuper en rien. Son contrat de développement, il le signe avec l'interface et non avec la classe.

C'est la responsabilité des autres développeurs de la classe `O2`, de faire en sorte que ce soit bien l'interface `IO2`, et aucune autre, qui soit implémentée. Les codes Java, C# et PHP 5 correspondant à ce diagramme sont présentés ci-après, répartis, dans les deux cas, et comme cela doit l'être idéalement, sur 5 fichiers séparés : les deux classes, les deux interfaces et la classe principale. Le développeur de la classe `O1` n'aura à manipuler que trois d'entre eux, au début, `IO1`, pour le donner aux autres, puis `IO2` pour savoir comment s'adresser à la classe `O2` et, plus longuement, `O1`, car c'est surtout pour cela qu'on le paie.

Code Java

Fichier 1 : IO1.java

```
 public interface IO1 {
public void jeTravaillePourO1();}
```

Fichier 2 : IO2.java

```
 public interface IO2 {
public void jeTravaillePourO2();}
```

Fichier 3 : O1.java

```
public class O1 implements IO1 {
  private IO2 unO2;

  public O1() { }
  public void setO2(IO2 unO2){
    this.unO2 = unO2;
  }
  public void jUtiliseO2() {
    System.out.println("j'utilise O2");
    unO2.jeTravaillePourO2();
  }
  public void jeTravaillePourO1(){
    System.out.println("je travaille pour O1");
    jImplementeLeServiceO1();
  }
  private void jImplementeLeServiceO1() {
    System.out.println("je suis prive dans O1");
  }
}
```

Fichier 4 : O2.java

```
public class O2 implements IO2 {
  private IO1 unO1;

  public O2 (IO1 unO1){
    this.unO1 = unO1;
  }
  public void jeTravaillePourO2() {
    System.out.println("je travaille pour O2");
    jImplementeLeServiceO2();
  }
  public void jUtiliseO1(){
    System.out.println("j'utilise O1");
    unO1.jeTravaillePourO1();
  }
  private void jImplementeLeServiceO2() {
    System.out.println("je suis prive dans O2");
  }
}
```

Fichier 5 : Principale.java

```
public class Principale {
  public static void main(String[] args) {
    O1 unO1 = new O1();
    O2 unO2 = new O2(unO1);
    unO1.setO2(unO2);
    unO1.jUtiliseO2();
    unO2.jUtiliseO1();
  }
}
```

Résultat

```
j'utilise O2
je travaille pour O2
je suis privé dans O2
j'utilise O1
je travaille pour O1
je suis privé dans O1
```

Rien de spécial dans ce code Java. Le code de la classe `O1` contient un référent vers l'interface `IO2` et *vice versa*. Aucun lien n'est établi syntaxiquement entre la classe `O1` et la classe `O2`. Ce n'est qu'à l'exécution, et de manière polymorphique, qu'un objet `O1` enverra explicitement un message vers un objet de la classe `O2`. Le compilateur n'y verra que du feu, et c'est comme cela que ça doit être. Tous les services rendus par la classe `O2` sont bien une implémentation de ceux prévus par l'interface.

Lors de l'exécution, l'interaction pourra se faire en toute tranquillité d'esprit, car rien de ce qui sera fait n'aura pas d'abord été prévu dans les interfaces, et vérifié par le compilateur. Pour ce dernier, les interfaces sont une garantie du respect des types, et en conséquence de la sécurité d'exécution. Notez qu'en Java, la seule compilation de la classe `Principale` entraîne la compilation de toutes les classes dont celles-ci dépendent, comme indiqué par le code. Les liens à la compilation se font, simplement, par le jeu de dénomination des fichiers et des classes.

Code C#

Fichier 1 : IO1.cs

```
 public interface IO1 {
void jeTravaillePourO1();}
```

À la différence de Java, point de mode d'accès dans la définition des méthodes, car ce mode d'accès ne peut être spécifié que lors de leur implémentation. Comme le seul mot-clé permis à ce niveau par Java est `public`, cela revient au même.

Fichier 2 : IO2.cs

```
 public interface IO2 {
void jeTravaillePourO2();}
```

Fichier 3 : O1.cs

```
using System;
public class O1 : IO1 {
  private IO2 unO2;

  public O1(){}
  public void setO2(IO2 unO2) {
    this.unO2 = unO2;
  }
  public void jUtiliseO2(){
    Console.WriteLine("j'utilise O2");
    unO2.jeTravaillePourO2();
  }
  public void jeTravaillePourO1(){
    Console.WriteLine("je travaille pour O1");
    jImplementeLeServiceO1();
  }
  private void jImplementeLeServiceO1(){
    Console.WriteLine("je suis prive dans O1");
  }
}
/* on ajoute une sous-classe qui doit à son tour implémenter l'interface */
public class FilsO1 : O1, IO1 /* l'interface n'est pas automatiquement héritée*/ {
  public new void jeTravaillePourO1() {
    Console.WriteLine("je travaille pour fils O1");
  }
}
```

Fichier 4 : O2.cs

```
public class O2 : IO2 {
  private IO1 unO1, unAutreO1;

  public O2 (IO1 unO1, IO1 unAutreO1) {
    this.unO1 = unO1;
    this.unAutreO1 = unAutreO1;
  }
  public void jeTravaillePourO2(){
    Console.WriteLine("je travaille pour O2");
    jImplementeLeServiceO2();
  }
  public void jUtiliseO1(){
    Console.WriteLine("j'utiliseO1");
    unO1.jeTravaillePourO1();
    unAutreO1.jeTravaillePourO1(); // c'est ici que la réimplantation de l'interface devient
     nécessaire
  }
  private void jImplementeLeServiceO2() {
    Console.WriteLine("je suis prive dans O2");
  }
}
```

Fichier 5 : Principale.cs

```
public class Principale {
  public static void Main() {
    O1 unO1 = new O1();
    FilsO1 unAutreO1 = new FilsO1();
    O2 unO2 = new O2(unO1, unAutreO1);
    unO1.setO2(unO2);
    unO1.jUtiliseO2();
    unO2.jUtiliseO1();
  }
}
```

Résultat

```
j’utilise O2
je travaille pour O2
je suis privé dans O2
j’utilise O1
je travaille pour O1
je suis privé dans O1
je travaille pour fils O1
```

À première vue, le code ressemble à s’y méprendre au code Java. Cependant, il sera toujours important en C# de se préoccuper des problèmes de typage statique et dynamique (et de l’emploi soigné des mots-clés : `abstract`, `virtual`, `new` et `override`), si l’héritage se propage vers le bas, et que de nouvelles classes héritent de la classe `O1`. C’est le cas ici avec la classe `FilsO1`. Comme seule la classe `O1` implémente l’interface, si nous ne forçons pas la classe `FilsO1` à ré-implémenter à son tour la même interface, par défaut et à cause du typage statique, le comportement de la méthode `jeTravaillePourO1()` sera celui de la superclasse `O1`, la première à implémenter l’interface et non pas celui de `FilsO1`, comme ce serait effectivement le cas en Java. N’oubliez pas que C# n’est pas polymorphique par défaut. Il a fait le choix de ne rien être par défaut et de vous obliger à préciser vos intentions.

C# a rendu cette pratique plus contraignante et moins intuitive que ne l’a fait Java, en évitant que vous vous laissiez guider par le seul fonctionnement par défaut (entièrement polymorphique en Java). Ainsi, cette ré-implémentation à plusieurs niveaux des interfaces pourrait conduire le programmeur C# à se trouver confronté à des problèmes de signatures de méthodes partagées entre plusieurs interfaces, qu’il ne pourra trancher qu’en précisant de quelle interface est issue la méthode (par une écriture comme « `IO1.jeTravaillePourO1() {….}` »).

À lire les développeurs de .Net, il semble que la raison essentielle de ce choix, ainsi que la présence du `new` dans la redéfinition des méthodes, soient liées à la possibilité de faire coexister dans un même code plusieurs versions des mêmes fonctionnalités. Tant que vous n’êtes pas convaincu du développement en cours de la méthode `jeTravaillePourO1()` dans la classe `FilsO1`, n’implémentez pas l’interface `IO1` dans celle-ci. Par défaut, c’est la version de la classe `O1` qui s’exécutera. Dès que vous êtes sûr de vous, vous pouvez ajouter l’implémentation de l’interface `IO1`. C’est automatiquement la nouvelle version qui sera exécutée, bien qu’il suffise d’un simple retrait pour revenir à la version précédente. Ainsi, les deux versions peuvent co-exister dans un même code. Cela peut aider le développeur à innover, puisqu’il a la certitude qu’une version ancienne continue à fonctionner comme roue de secours.

Si tous ces fichiers sont clairement tenus séparés, leur liaison, nécessaire lors de la compilation, recourt à une pratique moins souple qu’en Java, mais qui, en revanche, apparaîtra plus familière aux habitués du monde Microsoft. Pour qu’une classe, à la compilation, puisse en référer une autre, il faut d’abord la transformer en

une interface `.dll` (le même nom d'interface désignent ici deux types de fichier un peu différents, bien qu'ils servent tous deux à une forme de médiation entre d'autres fichiers, et c'est la raison pour laquelle, nous féminisons le second). L'instruction de compilation se transforme alors en :

```
csc /t:library /out :IO1.dll IO1.cs
```

pour faire de l'interface `IO1` un code récupérable à la compilation par les autres codes qui en dépendent. Ensuite, la compilation de la classe `O1` utilisant cette interface se fera par :

```
csc /r:IO1.dll /t:library O1.dll O1.cs
```

Et, ainsi de suite, « rebroussant » le fil des dépendances, jusqu'à la classe `Principale`, la seule à contenir le point de départ de l'exécution, c'est-à-dire la méthode `Main`. La classe `Principale` se compilera à l'aide de l'instruction :

```
csc /r:O1.dll /r:O2.dll /r:IO1.dll /r:IO2.dll Principale.cs
```

La connexion entre les interfaces et les fichiers « .dll » ne devraient pas surprendre les programmeurs habitués à l'environnement Microsoft. On retrouve ces interfaces dans le langage Visual Basic. On les retrouve également dans toute l'approche COM/OLE, si chère à Microsoft. Ainsi, en C#, même les structures peuvent hériter des interfaces et permettre à leur méthode de « s'extérioriser ». Ces interfaces serviront encore dans la nouvelle plate-forme .Net, en tant que pont entre les différents langages de programmation supportés par la plate-forme, tels que VB.Net, C#, C++, Python.Net et JScript. Ainsi, il sera possible d'utiliser du code C#, par l'entremise de son interface .dll, à l'intérieur d'une macro VB.Net. Les interfaces, ici, serviront non seulement à la médiation entre des classes écrites dans des environnements différents, mais élargiront cette médiation à des langages de programmation différents.

Code PHP 5

```
<html>
<head>
<title> Interfaces </title>
</head>
<body>
<h1> Interfaces </h1>
<br>

<?php

interface IO1 {
     public function jeTravaillePourO1();
}

interface IO2 {
     public function jeTravaillePourO2();
}

class O1 implements IO1 {
     private $unO2;

     public function __construct(){}

     public function setO2(IO2 $unO2) { // Il est possible de typer l'argument avec l'interface
          $this->unO2 = $unO2;
  }
```

```
  public function jUtilise02(){
    print("j'utilise 02 <br> \n");
    $this->un02->jeTravaillePour02();
     }

     public function jeTravaillePour01(){
          print("je travaille pour 01 <br> \n");
          self::jImplementeLeService01();
     }

     private function jImplementeLeService01(){
          print("je suis prive dans 01 <br> \n");
     }
}

class 02 implements I02 {
     private $un01;

     public function __construct (I01 $un01) { // Il est possible de typer l'argument avec l'interface
         $this->un01 = $un01;
     }

     public function jeTravaillePour02(){
          print("je travaille pour 02 <br> \n");
          self::jImplementeLeService02();
     }

     public function jUtilise01(){
          print("j'utilise01 <br> \n");
          $this->un01->jeTravaillePour01();
     }

     private function jImplementeLeService02() {
          print("je suis prive dans 02 <br> \n");
     }
}

$un01 = new 01();
$un02 = new 02($un01);
$un01->set02($un02);
$un01->jUtilise02();
$un02->jUtilise01();

?>

</body>
</html>
```

Nous installons tout le code dans un seul fichier .php. Rien de bien particulier à signaler. Le code est très proche du Java, sans nul besoin de typage statique, sauf lors du passage d'arguments où l'on peut, dans un souci de clarification, spécifier le type qui sera vérifié à l'exécution.

Les Interfaces dans UML 2

L'importance des interfaces dans le développement des logiciels OO est telle qu'UML a, dans sa deuxième version, enrichi son offre de symboles grapiques en matière d'interface. Ainsi, les liens entre les classes `O1`, `O2` et interfaces `IO1` et `IO2` décrits dans le chapitre précédent se représentent désormais, comme dans la figure 15-5, à l'aide d'un petit cercle lorsqu'il s'agit d'une implémentation d'interface et d'un « socket » (un arc de cercle comme un petit grippeur) lorsqu'il s'agit d'une utilisation d'interface. En général, le « socket interface » d'une classe se connecte sur le « cercle interface » d'une autre. C'est ainsi que les classes se connectent au mieux, par interfaces interposées. Ce nouveau symbole graphique s'est substitué au stéréotype « interface » apposé jusqu'alors sur les classes qui l'étaient. Cela nous permet également de comprendre la notion de stéréotype en UML, qui s'écrit entre guillemets, se rajoute sur n'importe quel élément graphique d'UML pour en particulariser l'usage et, en général, effecue la transition jusqu'à l'arrivée d'un nouveau symbole graphique.

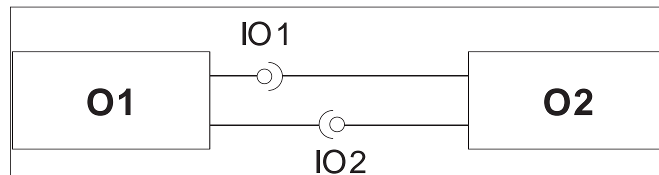


Figure 15-5
La classe O1 implémente l'interface IO1 et utilise l'interface IO2 et vice versa pour la classe O2.

Parmi d'autres additions, d'UML 2 figure le diagramme dit de « structure composite », dans lequel on trouve des éléments graphiques comme celui, extension de la figure précédente, représenté dans la figure 15-6.

Dans la figure 15-6, vous découvrez un composant logiciel et ses trois manières d'interagir respectivement avec ses trois « composants intelocuteurs ». Il peut s'agir d'une classe ou d'un composant plus important incluant plusieurs classes. Chaque interaction est symbolisée par un port comprenant une implémentation et une utilisation d'interface. L'implémentation correspond à la manière de ce composant de se « présenter » à son interlocuteur. La partie utilisatrice reprend les services que ce composant attend à son tour de la part de cet interlocuteur. Chaque interlocuteur se voit associé à un « port » à part. Il reprend le protocole de communication pour cet interlocuteur précis, chaque interlocuteur ayant le sien. À l'intérieur du composant, on découvre les parties fonctionnelles reliées à ces ports et qui concrétisent ces interfaces. Cette manière de représenter les composants logiciels par leur façon de s'imbriquer entre eux, les implémentations faisant office de « sortie » du composant et les utilisations des « entrées », ramène la conception logicielle à la réalisation d'un immense Lego ou puzzle, où l'on tente au mieux de construire un ensemble fonctionnel en assemblant entre elles des parties pré-existantes en fonction de leur « entrée-sortie ».

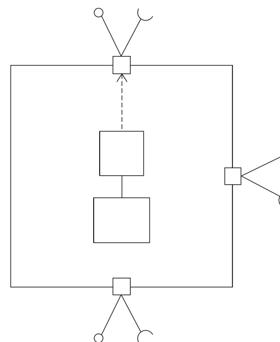


Figure 15-6
Partie d'un diagramme de « structure composite ».

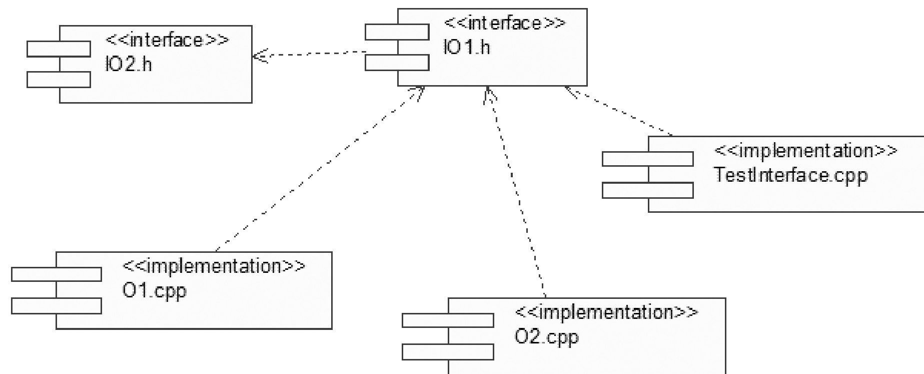
En C++ : fichiers .h et fichiers .cpp

Dans ce langage, le rôle de l'interface se réduit au plus essentiel de tous, c'est-à-dire une structure de code utilisée pour la médiation entre les différents acteurs d'un programme en cours de développement. Sa mise en œuvre est pourtant fondamentalement différente, car elle ne repose plus sur la pratique de l'héritage, mais plutôt sur la séparation dans l'écriture d'une classe entre un fichier dit d'interface, et portant l'extension `.h`, et un fichier dit d'implémentation, et portant l'extension `.cpp`.

L'interface n'est plus une addition syntaxique du langage, comme c'est le cas en Java et C#, mais elle répond plutôt à un mode d'organisation de l'application logicielle en un ensemble de fichiers présentant les classes aux autres, les `.h`, et de fichiers implémentant ces classes, les `.cpp`. L'interface n'est plus un élément essentiel de la syntaxe mais un type de fichier. Dans la version C++ des diagrammes UML, elle ne figurera plus dans le diagramme de classe, mais on la retrouvera dans le diagramme qui décrit l'organisation des fichiers de l'application : le diagramme de composant, comme la figure ci-dessous l'illustre.

Figure 15-7

Diagramme UML de composants associé au code C++ présenté plus haut.



Nous avons, jusqu'à présent, réalisé nos exemples de code C++ sans les faire précéder et les accompagner de fichiers d'interface. La raison en est simple, ils ne sont pas utiles à la compréhension première des briques de base de l'OO. Cependant, toute formation en C++ accorde, à juste titre, beaucoup d'importance à la décomposition, dans l'écriture des classes, entre fichiers d'interface et fichiers d'implémentation.

Excepté ce rôle de médiation, la définition du corps des méthodes dans le fichier d'implémentation, ou directement dans le fichier d'interface, n'est pas sans conséquence sur la taille et la vitesse d'exécution du code.

Implémenter une méthode dès la déclaration de la classe, comme nous l'avions fait jusqu'ici, plutôt que dans un fichier séparé, comme nous le ferons par la suite, conduit à une version différente de l'exécutable. Nous négligerons ces aspects, car notre ouvrage n'a pas pour vocation une compréhension exhaustive du C++. Cependant, tout développeur dans ce langage doit garder à l'esprit que cette séparation interface/implémentation, et l'endroit où se trouve déclarée l'implémentation des méthodes, est responsable d'un ensemble d'effets assez conséquents.

Nous allons reprendre l'application réalisée précédemment en Java, C# et PHP 5, en présentant et commentant à nouveau les cinq fichiers C++ qui supportent cette application.

Fichier 1 : IO1.h

```
#include "IO2.h"
class O1 {
private:
  O2 *unO2;
  void jImplementeLeServiceO1();
public:
  O1();
  void jeTravaillePourO1();
  void setO2(O2 *unO2);
  void jUtiliseO2();
};
```

Nous voyons qu'il n'est plus possible d'isoler dans l'interface un sous-ensemble des méthodes, les seules que nous voudrions rendre disponibles aux autres classes. Toutes les méthodes existant dans la classe devront être prévues dans l'interface. Néanmoins, l'interface se borne à n'en présenter que leur signature. C'est bien en cela qu'elle constitue, à nouveau, un partenaire essentiel à la décomposition de l'application logicielle. La classe O1 `include` l'interface `IO2.h`, `et` elle possède un attribut de type `O2`. C'est de cette manière qu'en C++ les classes interagiront entre elles. Chacune, dans le développement de son interface, « inclura » l'interface de celle qu'elle utilise, et en fera un attribut supplémentaire.

Fichier 2 : IO2.h

```
class O1;
class O2 {
private:
  O1 *unO1;
  void jImplementeLeServiceO2();
public:
  O2 (O1 *unO1);
  void jeTravaillePourO2();
  void jUtiliseO1();
};
```

Pour éviter des problèmes de récursivité sans fin, dus au simple fait que les deux classes se réfèrent mutuellement, nous ne procéderons pas ici à l'inclusion de l'interface `IO1.h`. En revanche, nous rappellerons simplement la classe `O1` au début du code. De nouveau, toutes les méthodes de la classe `O2` sont là, mais dans leur version la plus sobre, sans instruction. La classe `O2` possède, à son tour, un attribut de type `O1`.

Fichier 3 : O1.cpp

```
#include "stdafx.h"
#include "iostream.h"
#include "IO1.h"

O1::O1() { }
void O1::jeTravaillePourO1() {
  cout <<"je Travaille pour O1" << endl;
```

```
  jImplementeLeService01();
}
void O1::jImplementeLeService01() {
  cout << "je suis prive dans O1"<<endl;
}
void O1::setO2(O2 *unO2) {
  this->unO2 = unO2;
}
void O1::jUtiliseO2() {
  cout << "j'utilise O2" << endl;
  unO2->jeTravaillePourO2();
}
```

Comme nous le voyons, ce premier fichier d'implémentation, `O1.cpp`, doit commencer par inclure l'interface qu'il a pour mission d'implémenter. Ensuite, toutes les méthodes sont concrètement définies. Elles s'écrivent en faisant précéder leur signature de la classe à laquelle elles appartiennent.

Fichier 4 : O2.cpp

```
#include "stdafx.h"
#include "iostream.h"
#include "IO1.h"

O2::O2(O1 *unO1) {
  this->unO1 = unO1;
}
void O2::jUtiliseO1() {
  cout << "j'utilise O1" << endl;
  unO1->jeTravaillePourO1();
}
void O2::jImplementeLeServiceO2() {
  cout <<"je suis prive dans O2" << endl;
}
void O2::jeTravaillePourO2() {
  cout << "je Travaille pour O2" << endl;
  jImplementeLeServiceO2();
}
```

Rien de spécial à signaler, si ce n'est qu'il ne faut pas inclure `IO2.h`, vu que l'inclusion de la première interface s'en sera déjà occupée.

Fichier 5 : TestInterface.cpp

```
#include "stdafx.h"
#include "IO1.h"

int main(int argc, char* argv[]) {
  O1* unO1 = new O1();
  O2* unO2 = new O2(unO1);
  unO1->setO2(unO2);
```

```
    un01->jUtilise02();
    un02->jUtilise01();
    return 0;
}
```

Finalement, le `main`, détaché de toute classe, se borne à inclure seulement `I02.h`, pour les mêmes raisons que le fichier précédent.

Résultat

```
j'utilise 02
je travaille pour 02
je suis privé dans 02
j'utilise 01
je travaille pour 01
je suis privé dans 01
```

Séparation .h/ .cpp

En C++, la séparation des fichiers d'interface .h et d'implémentation .cpp est une autre façon d'améliorer la stabilité des logiciels, en forçant les développeurs, par l'organisation des fichiers (et non plus du code comme en Java, C# ou PHP 5), à désolidariser physiquement le catalogue des objets de leur accès direct.

Interfaces : du local à Internet

Par fichiers séparés

Ce qu'il y a de vraiment commun aux trois langages de programmation OO, c'est le besoin de séparer dans deux fichiers différents les signatures des méthodes de leur implémentation, ce qui est visible et accessible de l'extérieur selon la manière dont cela s'exécute de l'intérieur. Java, C# et PHP 5 permettent une extraction encore supplémentaire et plus fine, par le jeu de l'héritage. Dans ces langages, l'interface est re-solidarisée à l'implémentation par ce mécanisme d'héritage, alors qu'en C++ l'interface est `include` dans le fichier d'implémentation.

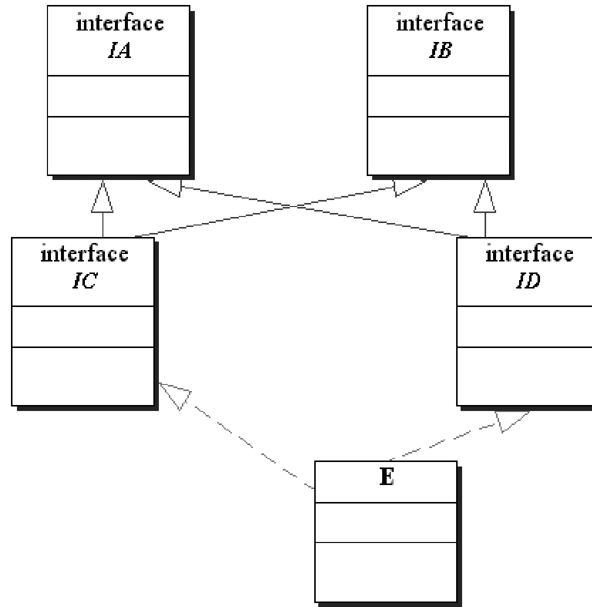
Nous allons généraliser dans le prochain chapitre cette interaction entre objets, par le truchement de leur interface, à Internet. Les interfaces sont à ce point suffisantes à la communication entre objets que seuls les fichiers qui les contiennent devront être installés sur les ordinateurs séparés, mais appelés à communiquer. La véritable implémentation des services sera non seulement encapsulée dans les fichiers classes, mais également dans des ordinateurs séparés, ordinateurs simplement connectés par Internet et son protocole de communication : TCP/IP.

Exercices

Exercice 15.1

Écrivez en Java ou en C# le code correspondant à ce diagramme de classe, composé de quatre interfaces et d'une classe :

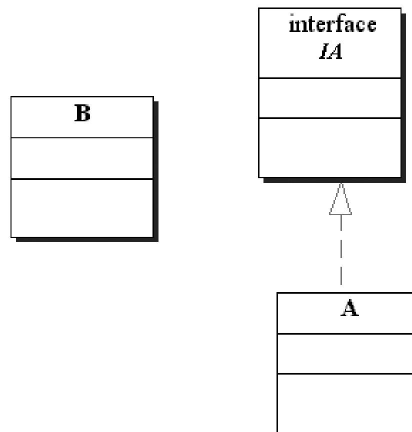
Figure 15-7



Exercice 15.2

Si la classe B cherche à interagir avec la classe A, comment allez-vous représenter la flèche d'association dirigée dans le diagramme de classe ci-après ?

Figure 15-8



Exercice 15.3

Expliquez en quoi l'utilisation des interfaces rend possible le multihéritage.

Exercice 15.4

Décomposez en un fichier d'interface `.h` et un fichier d'implémentation `.cpp`, le seul fichier d'implémentation C++ suivant :

```
class A {
  private:
    int a1, a2;
    bool b1, b2;

    int faireA() {
      return a1 + a2;
    }
  public:
    A(int a1, int a2) {
      this->a1 = a1;
      this->a2 = a2;
      b1 = false;
    }
    void faire2A(int c) {
      faireA() + c;
    }
};
```

Exercice 15.5

Lors de la compilation du code Java ci-après, deux erreurs sont signalées. Lesquelles ?

```
interface IA {
  private int faireA();
  public void faire2A();
  public int faire3A();
}
class A implements IA {
  public int faireA() {
    return 2;
  }
  public void faire2A() {
    System.out.println(faireA());
  }
}
public class Exercice5 {
  public static void main(String[] args) {
    A unA = new A();
    unA.faire2A();
  }
}
```

Exercice 15.6

Lors de la compilation du code C# suivant, trois erreurs sont signalées. Lesquelles ?

```
using System;
interface IA {
  public int faireA();
  public void faire2A();
}
class B {
  public int faireB() {
    return 3;
  }
}
class A : IA, B {
  public int faireA() {
    return 2;
  }
  public void faire2A() {
    Console.WriteLine(faireA());
  }
}
public class Exercice6 {
  public static void Main() {
    A unA = new A();
    unA.faire2A();
  }
}
```

Exercice 15.7

Dans quel cas de figure une classe abstraite héritera-t-elle d'une interface ?

Exercice 15.8

Pourquoi en C++, au contraire de C# et de Java, le fichier implémentation contiendra-t-il le même nombre ou moins de méthodes que le fichier interface ?

Exercice 15.9

En C# et en Java, un attribut peut-il posséder en tant que type une interface ? Si oui, est-ce là une bonne pratique ?

Exercice 15.10

Expliquez les trois pratiques de compilation liées à la présence des interfaces, et pourquoi la pratique de Java apparaît comme la moins lourde à mettre en œuvre.

16

Distribution gratuite d'objets : pour services rendus sur le réseau

En s'interrogeant tout d'abord sur les raisons de leur utilisation, ce chapitre introduit à la pratique des applications informatiques distribuées, applications réalisées par le biais d'objets distribués, s'activant sur des ordinateurs séparés et communiquant à travers la couche physique : Internet ou un niveau sémantique au-dessus : le Web. Différentes implémentations de ces objets distribués seront évoquées et expérimentées, comme RMI, Corba, Jini et les services Web.

Sommaire : Objets distribués : pourquoi ? — Invocation statique — RMI — Corba — Jini — Invocation dynamique — Services Web

Candidus — Et si nous en venions à l'essentiel… Tous ces objets représentent bien plus qu'un nouveau type d'organisation. J'ai le sentiment que c'est même presque une révolution des modes de pensée.

Doctus — Je suis très content de t'entendre dire ça Il était temps que les informaticiens se mettent à suivre l'exemple des constructeurs de matériel informatique. Depuis un certain temps, ces derniers nous en font une tous les deux ans en matière de performance et de nouvelles technologies.

Cand. — Alors que les programmeurs se contentent de leurs bons vieux « if then else » !

Doc. — Internet fait la démonstration tous les jours de ce qu'on peut faire en matière de logiciels distribués. Nos objets ne sont rien d'autre que des moyens modernes pour répartir les tâches de manière démocratique, à travers toute la planète. C'est l'OO à l'heure de la mondialisation, n'en déplaise aux alterégOO.

Cand. — Une démocratie à la fois utopique et bien réelle alors ! Dans laquelle chacun a voix au chapitre pour exercer son droit à la liberté d'expression mais où les responsabilités devront être pleinement assumées !

Doc. — Il s'agit là d'un aspect très fort de notre défi. Cette responsabilisation n'est possible que si elle est basée sur de réelles compétences.

Cand. — Il faudra donc veiller à la bonne distribution des rôles en s'assurant que chacun disposera de tout le nécessaire pour effectuer son travail. Chaque objet devra également pouvoir communiquer avec tous les spécialistes qu'il lui faudra connaître pour compléter son savoir-faire.

Doc. — Les langages actuels sont assez avancés pour exploiter tout le travail réalisé avant l'OO. Ce qu'il reste à faire ne concerne que l'aspect d'ouverture des programmes existants pour les faire communiquer, afin qu'on puisse s'en servir, dans la situation et le moment voulus.

Objets distribués sur le réseau : pourquoi ?

Faire d'Internet un ordinateur géant

La possibilité d'utiliser des objets qui s'exécutent, indépendamment, sur des processeurs séparés, mais tout en continuant à s'envoyer des messages, cette fois-ci à travers Internet, se justifie par un ensemble d'avantages, pour la plupart liés à l'exploitation des réseaux informatiques en général. Le premier d'entre eux est d'accroître les ressources computationnelles mises à disposition. On connaît le fameux slogan de Sun : « The computer is the network » (la traduction ne nous paraît pas indispensable).

Cela reste évidemment le cas, tant que le coût des communications entre les objets (par exemple la durée d'une communication multipliée par le nombre de communications) est inférieur à l'épargne effectuée par la répartition de l'exécution sur différents processeurs. Internet est un immense ordinateur, le plus grand et le plus puissant, pour autant que l'on arrive à paralléliser efficacement l'application logicielle qui nous intéresse sur tous les nœuds qui le constituent.

C'est avec une mélancolie certaine que les utilisateurs gourmands en temps calcul rêvent à tous ces ordinateurs dormants qui, quand ils se réveillent, ne tournent qu'à 10 % de leur capacité de temps calcul, et ce quand ils tournent vraiment, entre deux e-mails et deux téléchargements de la photo osée de la dernière vedette d'un reality show.

> L'exemple le plus connu d'une telle utilisation des ressources Internet est le projet « seti@home », où tous les ordinateurs du réseau qui le désirent dédient une partie de leurs ressources (temps calcul et mémoire) à la recherche de motifs tangibles dans des signaux radio perçus dans l'univers entier, bouts de signal révélateurs d'une éventuelle présence intelligente dans l'univers (en dehors du bureau des auteurs, bien entendu). Les extraterrestres auraient démarré un projet semblable, mais leurs ordinateurs sont, depuis, pris dans une boucle infinie et à cours de ressources, dans l'analyse des écrits de Heidegger, des chansons de Léo Ferré, des films de Greenaway, des tableaux de Pollock et des musiques de Boulez.
>
> « seti@home » est très facilement parallélisable car chaque ordinateur jouant le jeu, ne s'occupe que d'une partie du ciel, tous font la même chose mais chacun se limite à une partie des données. C'est également le cas dans de nombreux projets bioinformatiques comme le séquençage de motifs d'ADN, le repliement des protéines ou la prédiction climatique, eux aussi facilement parallélisables. Citons ainsi le projet Folding@home de l'Université de Stanford partageant sur tous les ordinateurs qui y participent la recherche des caractéristiques des protéines liées à des maladies telles que Alzheimer ou de nombreux cancers.

La motivation essentielle des nouveaux projets d'informatique distribuée comme le « Grid Computing » qui répartit sur tous les ordinateurs d'Internet qui acceptent d'y consacrer un peu de leurs ressources des tâches très exigeantes en mémoire et temps calcul est que, quel que soit l'accroissement des performances informatiques (comme la loi de Moore nous l'indique), cet accroissement sera toujours négligeable par rapport à celui des performances du réseau dans sa totalité qui, non seulement bénéficie de l'accélération de chacun de ses membres, mais également de l'accroissement du nombre de ceux-ci dans le réseau. Le mélange du « Grid Computing » et des services Web devrait connaître une évolution très naturelle.

Répartition des données

Une autre raison d'utiliser Internet pour la communication entre objets tient au simple fait que les objets doivent parfois s'exécuter sur des processeurs distribués, car les données à manipuler sont elles-mêmes réparties dans la mémoire connectée à ces processeurs. C'est souvent le cas de la lecture ou de la mise à jour de base de données, laquelle ne peut s'effectuer que sur le serveur où cette base est installée. Déplacer toute la base serait incommensurablement plus coûteux que simplement déplacer l'objet, qui nécessite quelques informations en provenance de cette base. La plupart des applications distribuées, appliquées au commerce électronique, sont de ce type. On les dit « 2-tier », lorsqu'un client interagit, *via* son navigateur Internet, directement avec le serveur de

la base de données. C'est plutôt rare, car les connexions directes sur une base de données sont lentes, coûteuses et concurrentielles.

De ce fait, on rajoute le plus souvent quelques processeurs intermédiaires ; l'application est dite alors « 3-tier ou multi-tier ». Ces processeurs récoltent les requêtes clients et, avant d'entamer une interaction avec la base de données (interaction lente et à réaliser avec parcimonie), vérifient les requêtes (pour cela, ils peuvent s'embarquer dans quelques interactions supplémentaires avec le client), compilent ces requêtes, les transforment, les homogénéisent puis, enfin, les transmettent à la base de données.

Tous ces allers-retours entre les clients, les serveurs intermédiaires et les bases de données, peuvent se faire idéalement avec les objets distribués, et sont d'ailleurs très fréquemment réalisés à partir du protocole propre à Java (dû au positionnement stratégique de Java pour les applications Web), « RMI », étudié à la prochaine section. Ici, les objets sont distribués, purement et simplement, car les données qu'ils doivent traiter, pour des raisons historiques, économiques, stratégiques ou de confidentialité, le sont également.

Répartition des utilisateurs et des responsables

Au contraire des lofteurs ou des star académitiens, tous les utilisateurs de l'informatique ne partagent pas un même lieu géographique, et ne l'utilisent pas pour les mêmes raisons. Ils sont tout autant distribués que l'est leur machine. Et leur mobilité est devenue celle de leur portable. L'existence des objets distribués conduit à faciliter l'intégration de services rendus par des entreprises commerciales distinctes mais, éventuellement, complémentaires. À ce titre, on parle souvent et de plus en plus, d'architecture informatique orientée service (SOA).

Par exemple, l'organisation d'un voyage implique des services pour la prise en charge du déplacement, d'autres pour celle du séjour, d'autres encore pour celle des assurances, etc. Une agence de voyage pourra plus simplement interagir avec ces différentes entreprises, de manière à centraliser aisément toute la gestion du voyage. Chaque utilisateur peut collaborer avec les autres en déléguant respectivement leurs tâches à des objets distribués. On entend souvent parler sur Internet d'agents « intelligents », capables de prises de décision, d'achats, de ventes, au profit de son « maître ». Ces agents se trouvent le plus souvent incarnés dans des objets distribués. C'est aussi derrière ce type d'informatique que courent les services Web : permettre à moindre frais d'intégrer et de faire collaborer des compétences distribuées géographiquement. Vous achetez un disque et le magasin dans lequel vous vous trouvez vous permet de réserver des places pour le concert de l'artiste auteur du disque que vous achetez. L'achat des logiciels informatiques devrait être lentement mais sûrement remplacé par la location de ces mêmes logiciels le temps nécessaire à leur utilisation.

> On assiste au développement d'une informatique éclatée, distribuée sur le réseau. Les programmes, qui s'exécutent sur des machines distantes se sollicitent mutuellement au fur et à mesure de leur exécution. Les deux petits exemples qui suivent permettront aisément de comprendre ce nouvel emploi du Web, à portée de souris aujourd'hui. Si vous utilisez un traitement de texte et désirez corriger votre texte à l'aide du meilleur correcteur orthographique existant sur le marché, vous êtes limité au seul programme disponible dans le traitement de texte que vous utilisez. Demain, lorsque vous solliciterez dans le menu de votre traitement de texte la fonction « correcteur orthographique », de nombreuses options s'offriront à vous, différenciées par leur temps d'utilisation, leur prix, leur qualité, etc. Il vous sera alors possible de choisir celle qui vous convient. Automatiquement, ce choix entraînera l'empaquetage de votre texte, de manière à le faire circuler sur le Web, à l'acheminer vers le correcteur orthographique en question (à l'Académie française ou à Oxford), qui le corrigera et vous le renverra dans sa version corrigée. Un système de facturation totalement intégré vous permettra de ne payer que ce que vous aurez dépensé (accès et temps de correction). Vous ne serez plus obligé d'acheter et de mettre constamment à jour sur votre ordinateur un correcteur unique. Ce scénario est évidemment extensible à tous les logiciels utilisés, y compris le traitement de texte de départ, dont nous supposons que vous êtes en train de vous énerver à installer la version 2010, n'en déplaise à Microsoft.

Peer-to-peer

Le modèle « peer-to-peer », dont un logiciel comme Napster fut à l'origine (mais sa totale vulnérabilité en raison de la présence d'un serveur central l'a fait disparaître au profit d'autres implantations comme Gnutella ou Kazaa dont les serveurs sont plus distribués) est très représentatif d'une vision idéalisée d'Internet, « new age » et « collectiviste », défendue par beaucoup. Tout ordinateur devrait être, à la fois, serveur de données et client de ces mêmes serveurs, en évitant au maximum de passer par des ordinateurs pivots, jouant un rôle trop stratégique de concentrateurs ou de médiateurs.

Si les réseaux informatiques ont eu et continuent à avoir cet extraordinaire impact psychologique et économique sur le fonctionnement de nos sociétés, c'est que leur diffusion s'accompagne, tout en les amplifiant, de deux phénomènes d'importance croissante : la communication et la dématérialisation de ce que l'on communique. De plus, ces deux phénomènes s'amplifient mutuellement : l'accroissement des communications pousse à la dématérialisation des supports d'échange (voyez la musique, voyez les films) et l'accroissement de cette dématérialisation banalise la communication de ceux-ci (voyez les musées Internet ou le commerce électronique en général).

La dématérialisation des supports d'échange est une tendance constante, qui nous accompagne depuis l'origine des temps. Le troc, qui consistait à échanger des produits mais exigeait de multiples relations interpersonnelles, fit place à la monnaie métallique, qui facilitait l'échange, parce qu'elle désolidarisait la transaction des produits échangés. De nos jours, cette monnaie matérielle est de plus en plus souvent remplacée par des transactions électroniques, ce qui accroît la distance dans le temps et dans l'espace entre acheteurs et vendeurs. Pièces, billets, chèques s'effacent chaque jour un peu plus au profit des cartes de crédit et des sites Web. Cela montre que l'objet de l'échange survit très souvent et très bien aux mutations que subit le support de l'échange.

Le besoin d'échanger demeure tandis que le médiateur se métamorphose. L'informatique a largement accompagné cette tendance à la dématérialisation, avant d'en devenir un vecteur et un accélérateur prépondérant. Ainsi, en matière d'économie, nous vivons une évolution conséquente qui nous fait passer des échanges matériels de biens et de marchés à des relations fondées sur le seul accès dématérialisé à un bien, accès limité dans le temps et facilité par les réseaux. De nombreux économistes pensent que nous paierons dans l'avenir moins pour le transfert de propriété d'un bien dans l'espace que pour le « flux d'expérience » auquel nous aurons accès dans le temps. Par exemple, nous n'achèterons plus un DVD mais nous paierons le visionnage d'un film, téléchargé en direct à partir d'une agence de location vidéo. Le marketing des objets les plus matériels risque lui aussi d'être modifié par cette pratique nouvelle. En effet, nous n'achèterons plus une voiture, mais nous la louerons le temps d'un voyage, tout comme vous le faites aujourd'hui lorsque vous louez une place dans un avion, mais probablement pour d'autres raisons.

Cette parfaite mise à plat du réseau Internet, ce projet par trop égalitaire, deviendrait, ce faisant, un réseau de communication fantastique pour des objets voués à se rendre mutuellement des services à travers le réseau. « Passe-moi ton texte que je te le corrige avec le merveilleux correcteur sémantique dont j'ai fait l'acquisition récemment, et j'en profite pour t'envoyer une photo à retoucher avec ton merveilleux système de traitement d'images, et puis, surtout, n'hésite pas à m'envoyer ta dernière production musicale dont mon merveilleux compositeur automatique fera du Mozart. »

Il ne s'agit pas ici de simples transferts de fichiers, mais plutôt de déclencher des applications à distance, en leur transmettant en paramètres les données sur lesquelles ces applications vont devoir opérer. C'est toute la différence entre un envoi de message, qui est un ordre d'exécution, et un envoi de données. Les autoroutes de l'information se transformeront en des réseaux de petits boulots, où l'on troque des services plutôt que des biens, et où les 35 heures seront très difficiles à faire respecter. Sun a beaucoup misé sur cette évolution et favorise le développement d'une plate-forme informatique nommée « JXTA », qui devrait faciliter grandement le développement, par chacun, d'applications « peer-to-peer ».

L'informatique ubiquitaire

Une autre évolution, plus récente, veut que les ordinateurs, tels qu'on les connaît, c'est-à-dire une boîte grise ou fluo, sur la table ou sur les genoux, soient de moins en moins les seules machines à vouloir communiquer. Dans un avenir très proche, dans le monde de l'Internet sans fil, n'importe quelle machine, dotée d'un processeur, pourrait chercher à communiquer avec n'importe quelle autre. C'est ce qu'on appelle aujourd'hui l'informatique ubiquitaire ou disparaissante, à votre convenance. Bien que contradictoires, les deux adjectifs décrivent des réalités qui se conditionnent. C'est parce qu'il y a des ordinateurs partout qu'il n'est plus opportun de parler d'ordinateur, un peu comme les chauves, qui sont les seuls à parler de cheveux (oui, on sait, comme analogie, c'est un peu tiré par les cheveux…).

Rappelez-vous le premier chapitre (si vous y parvenez, c'est si loin…). La voiture que nous y décrivions pourrait réellement envoyer un message au feu rouge, comme « passe au vert, j'arrive ». La brosse à dents pourrait demander au percolateur de lancer le café, ou l'inverse. Le frigo pourrait commander au supermarché voisin le renouvellement de ses denrées périmées. Votre appareil photo pourrait envoyer la prise de vue passionnante du coucher de soleil que vous venez de faire, sur la télévision de votre fils, en train de retransmettre le match Allemagne-Brésil.

Imaginez-vous dans un pays lointain avec votre appareil photo numérique. Vous souhaitez imprimer avec la meilleure qualité possible la photo que vous venez de faire et n'avez aucune imprimante le permettant. Dans votre appareil photo un petit menu affiche la liste des boutiques qui peuvent s'occuper de cette impression dans un rayon d'un kilomètre. Vous en choisissez une, là encore selon certains critères : prix, proximité, qualité. Une fois votre choix effectué, un seul clic suffit à lancer l'impression. Vous n'aurez plus alors qu'à aller chercher votre photo imprimée dans la boutique en question. La facturation s'effectuera, à nouveau, le plus automatiquement et simplement qui soit sous forme électronique. Vous en rêviez… Les objets distribués l'ont fait. Nous reviendrons sur cet exemple dans la suite du chapitre. Rien n'est vraiment fictif dans tout cela, la technologie le permet ; il reste à rendre toutes ces idées utiles, puis, un jour, indispensables.

Robustesse

Un dernier avantage à la répartition à travers le réseau d'une application informatique tient à la robustesse de cette application face aux pannes de machines, si le même traitement s'effectue, en toute redondance, sur plusieurs ordinateurs à la fois. L'ambition idéale d'Internet était de renforcer, par ses multiples ordinateurs et ses multiples liaisons, la fiabilité du système informatique. Plus un réseau contient de nœuds et de connexions, plus la fiabilité de ses communications est assurée. En clair, la multiplication des ressources mises à disposition, pour un projet computationnel unique, garantit une quasi parfaite fiabilité dans l'exécution de ce projet.

RMI (Remote Method Invocation)

Très peu de modifications devront être apportées aux fichiers Java utilisés dans le chapitre précédent, dès lors que nous souhaitons simplement « migrer » cette même application sur Internet. Nous débutons la pratique des objets distribués par la méthodologie RMI car, d'une certaine manière, c'est elle qui satisfait le mieux l'ambition poursuivie par toutes les autres : rendre transparente la distribution géographique des objets s'échangeant des messages pour le programmeur, ce dernier ayant la possibilté de programmer « distribué » exactement comme il programme en local. Bien qu'elle partage l'ambition de rendre le développement d'applications globales presque aussi immédiat que le développement d'applications locales, c'est la force de la technologie RMI (Remote Method Invocation), par rapport à la technologie Corba, que nous discuterons dans la suite, de passer, effectivement, le plus facilement,

d’une application Java purement locale à une version distribuée de cette même application. Les cinq fichiers Java nécessaires, comme dans le chapitre précédent, sont indiqués ci-après.

Nous allons détailler uniquement les additions qui font de cette application, précédemment locale, une application Internet. Pour les différencier du cas précédent, nous avons simplement ajouté « RMI » à la fin du nom des classes et interfaces concernées. Comme nous travaillons à travers le réseau Internet, des problèmes sont à prévoir, panne de réseau, connexion impossible, que Java nous force, de fait, à anticiper. En conséquence de quoi, les seules additions requises concerneront surtout des mécanismes de gestion d’exception. Nous allons développer les côtés serveur et client de la communication, bien que nous sachions qu’en matière d’objets distribués, ces deux appellations sont parfaitement interchangeables.

Côté serveur

Fichier 1 : IO2RMI.java – Définition de l’interface

```
public interface IO2RMI extends java.rmi.Remote {
  public void jeTravaillePourO2() throws java.rmi.RemoteException;
  public String jeRenvoieUnString() throws java.rmi.RemoteException;
}
```

Nous avions anticipé dans les chapitres précédents l’importance prise par les « interfaces » dans la réalisation des applications distribuées. Il est en effet plus important encore, lorsque les programmeurs se trouvent largement séparés aussi bien dans l’espace que dans le temps, de baser leur communication sur un protocole d’échange minimal, leur laissant les mains libres pour d’éventuelles transformations dans les applications dont ils ont la charge. Nous découvrons dans la suite, leur exploitation dans ce cadre précis d’objets distribués sur des processeurs distincts. Pour indiquer qu’elle sera utilisée à distance, l’interface doit hériter de la classe `java.rmi.Remote`. Nous prévoyons deux services dans cette interface, le premier se borne à écrire sur le serveur, alors que le second transmet un `String` que le client doit recevoir. Il faut prévoir un éventuel déclenchement de l’exception `java.rmi.RemoteException`, quand on enverra le message prévu par cette signature. Bien des choses pourraient se passer sur le réseau, rendant l’envoi du message impossible. Le message, dès l’écriture de sa signature, doit également contenir les exceptions auxquelles il peut donner lieu. Les problèmes qui peuvent se produire, et qu’il est nécessaire de prévoir, font partie de la signature du message.

Fichier 2 : O2RMI.java – Définition de la classe, implémentation de l’interface par la classe O2RMI

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class O2RMI extends UnicastRemoteObject implements IO2RMI {
  public O2RMI() throws RemoteException {
    super();
    /*System.setSecurityManager(new RMISecurityManager()); */
    System.out.println("Je mets mon objet sur le net");
    try {
      Naming.rebind("unObjetO2", this); /* enregistrement de l'objet */
    } catch (Exception e) {System.exit(0);}
    System.out.println("C'est fait");
  }
```

```
  public void jeTravaillePourO2() { /* implémentation du premier service */
    System.out.println("je travaille pour O2");
    jImplementeLeServiceO2();
  }
  public String jeRenvoieUnString() { /* implémentation du deuxième service */
    return "un bonjour en provenance d'O2";
  }
  private void jImplementeLeServiceO2() {
    System.out.println("je suis prive dans O2");
  }
}
```

La classe qui implémente les deux messages possibles (et donc l'interface contenant leur signature) doit également hériter de `UnicastRemoteObject`, si elle désire redéfinir le comportement à distance des méthodes, telles `equals` ou `toString`, qu'il peut être important de repenser vu la distribution des objets sur le réseau. La principale différence avec une application locale se situe dans le constructeur de la classe `O2RMI`.

Il faut maintenant prévoir que l'objet en question, qui exécutera les services du côté serveur, soit référencé par un nom. Il faut installer cet objet sur le Web avec un nom symbolique, ici `unObjetO2`, au moyen duquel tout client pourra y faire appel. Ce nom sera installé dans un « registre » que tout client pourra consulter. Lors de la déclaration du nom du référent dans le « registre », par l'opération `Naming.rebind()`, un port, autre que celui spécifié par défaut (this), peut être précisé.

Un service de sécurité peut également être activé (ici, mis en commentaire) qui, souvent, se limite à celui par défaut. On admettra, sans s'en préoccuper davantage, que toute interaction de type réseau ne va pas sans un souci sécuritaire (dissimulation du contenu du message, identification de l'expéditeur ou du destinataire…) qu'il faudra prendre en charge. Les deux messages à exécuter, `jeTravaillePourO2()` et `jeRenvoieUnString()`, sont là, mais cette fois avec leur implémentation. Même si leur mode d'exécution se fera à distance plutôt que localement, rien ne change vraiment dans l'écriture de leur « envoi ».

Fichier 3 : PrincipaleServeurRMI.java – Fichier final côté serveur

```
public class PrincipaleServeurRMI {
  public static void main(String[] args) {
    try {
      new O2RMI();
    } catch (Exception e) {System.exit(0);}
  }
}
```

Il nous faut un exécutable principal côté serveur, mais rien de particulier n'est à signaler dans le code. Il se borne à déclencher toute la mise en disponibilité de l'objet côté serveur, en appelant simplement le constructeur de la classe `O2RMI`, afin de créer l'objet qui sera installé sur Internet.

Côté client

Fichier 4 : O1RMI.java – Fichier réalisant l'appel des services côté client

```
import java.rmi.*;
public class O1RMI  {
  IO2RMI unObjetO2 = null;
```

```
  public O1RMI(){
    System.out.println("je vais chercher l'objet sur le net");
    try {
      /* il faut retrouver l'objet sur lequel déclencher les services */
      unObjetO2 = (IO2RMI)Naming.lookup("unObjetO2");
    }
    catch (Exception e) {
      System.out.println("ça marche pas " + e);
      System.exit(0);}
      System.out.println("c'est fait");
  }
  public void jUtiliseO2() {
    System.out.println("j'utilise O2");

    try /* on envoie les deux messages */ {
      unObjetO2.jeTravaillePourO2();
      System.out.println(unObjetO2.jeRenvoieUnString());
    } catch (Exception e) {
      System.out.println("ça marche pas" + e);
      System.exit(0);
    }

    System.out.println("c'est fait");
  }
}
```

Afin de pouvoir envoyer son message, le client doit, avant tout, identifier son destinataire. Il le fait en utilisant le même « registre » que le serveur. Il cherche sur ce « registre » l'objet qui correspond au nom `unObjetO2`, qui est en effet le nom que nous lui avons donné côté serveur. Il est important, à ce stade, de compléter le nom de l'objet par l'adresse Internet de l'ordinateur sur lequel il se trouve, ainsi que le port de communication qui est dédié à cette interaction. En général, cette même opération s'écrira plutôt de la manière suivante :

```
Naming.lookup (" rmi ://iridia.ulb.ac.be :1234/unObjetO2 ")
```

`iridia.ulb.ac.be` étant l'adresse Internet du serveur en question, et 1234 le port de communication. Nous ne l'avons pas fait ici, puisque nous exécutons toute l'application en local.

Un objet dans chaque port

Un port est un « canal de sortie » dédié à une communication d'un ordinateur donné avec le monde extérieur. Chaque communication avec chaque interlocuteur doit faire l'objet d'un port distinct et dédié à cette seule communication. Du côté du serveur, l'objet est également « présent » et « à l'écoute » sur ce même port.

À ce stade, une véritable distribution du service sur Internet ne demande en fait qu'une modification de l'adresse dans le code du client, les deux envois de message sur `unObjetO2` se déroulant exactement comme si cette application ne fonctionnait qu'en local. Comme discuté dans le chapitre précédent, nous voyons que le client exige de connaître l'interface `IO2RMI`, qui sert effectivement de médiateur entre les deux acteurs : du côté serveur, on l'implémente, du côté client, on l'utilise.

L'interface encore

La structure syntaxique d'interface permet une véritable séparation physique des développements informatiques, non seulement entre les programmeurs mais également entre les machines exécutant les programmes. La seule information en provenance du serveur dont le client dispose est l'interface des services rendus par celui-ci afin qu'il puisse compiler.

Fichier 5 : PrincipaleClientRMI.java – Finalement le fichier principal côté client

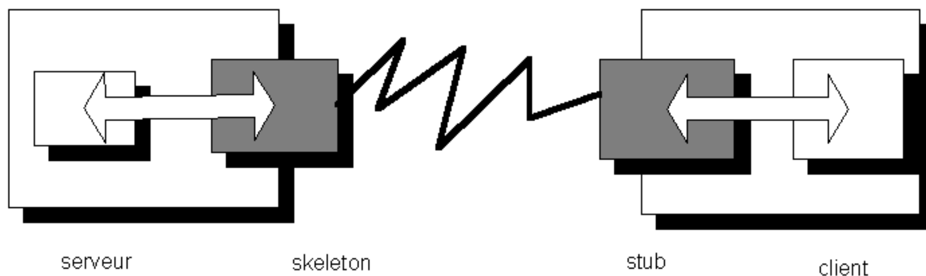
```
public class PrincipaleClientRMI {
  public static void main (String args[]) {
    O1RMI unObjet01 = new O1RMI();
    unObjet01.jUtilise02();
  }
}
```

Rien à signaler.

RMIC : stub et skeleton

Déclencher une application distribuée, *via* le protocole RMI, est à peine plus exigeant qu’une simple application locale. Une étape additionnelle nécessaire, dans le cas d’un processus dit « invocation statique », est celle de la création du « stub » et du « skeleton », par la ligne de commande `rmic O2RMI`, à exécuter sur le fichier compilé en Java (`.class`) de la classe qui effectue le service, côté serveur. Suite à cette instruction, deux nouveaux fichiers s’ajoutent, l’un sur l’ordinateur côté client, le « stub » : `O2RMI_Stub.class`, et l’autre sur l’ordinateur côté serveur, le « skeleton » : `O2RMI_Skel.class`. À quoi servent ces deux fichiers additionnels, générés automatiquement par Java ?

Comme la figure ci-après le montre, ils sont indispensables à l’envoi du message. En substance, le stub « empaquette » le message côté client, et s’occupe, en pratique, de la transmission du message vers le serveur. De son côté, le skeleton « dépaquette » le message côté serveur, l’active sur l’objet destinataire, « empaquette » les résultats (le return du message s’il y en a un) et transmet ce résultat au client. Le stub, encore lui, sera chargé de récupérer et de « dépaqueter » le return, afin de permettre au client d’aller de l’avant avec son exécution. C’est la présence de ces deux exécutables additionnels qui permet aux développeurs d’écrire une application distribuée, aussi simplement qu’une application locale. Dans un processus d’invocation statique, la plus grosse partie du boulot est ainsi faite, gracieusement, par le stub et le skeleton. Dans les dernières versions de Java, le skeleton n’est plus nécessaire et ne sera plus généré par la commande `rmic`. Le serveur découvre par un mécanime de « réflexion », c’est-à-dire directement à partir de l’objet serveur, les méthodes qui sont accessibles sur celui-là. À cet allègement près, rien ne change dans la mise au point de la solution RMI.



igure 16-1

Les rôles joués par le stub et le skeleton.

Invocation statique versus invocation dynamique

L'exemple de RMI que nous présentons pour l'instant, de même que son équivalent Corba, réalise cet envoi de message à travers Internet grâce à un mécanisme d'invocation statique pour lequel un stub reste indispensable afin de jouer la doublure du serveur du côté client. En substance, lors de l'invocation statique, plus exigeante mais plus simple à mettre en œuvre que l'invocation dynamique, le client doit disposer d'une copie de l'interface du serveur (pour compiler) et du stub, également en provenance du serveur (pour pouvoir s'exécuter). En revanche, l'invocation dynamique ne nécessite plus la présence de cette doublure et de ce stub et permet de découvrir sur le moment et « à chaud » les services dont un premier objet a besoin en provenance de l'interface d'un deuxième. Nous reparlerons de l'invocation dynamique.

Lors d'une invocation statique, le stub serait comme une doublure ou un substitut du serveur pour le client. En son absence, le programme ne pourrait s'exécuter côté client, car il n'y aurait personne à qui envoyer le message. Par exemple, il est important que les paramètres du message soient empaquetés de telle manière que le skeleton puisse aisément les dépaqueter. L'un ne va donc pas sans l'autre. Le skeleton apparaît comme le dual du stub côté serveur, faisant, lui, office d'une doublure ou d'un substitut du client.

La présence de ces deux modules, stub et skeleton (répétons que le skeleton a disparu des dernières versions de Java), est générale à toutes les applications distribuées qui fonctionnent par invocation statique. On les retrouvera de manière très semblable dans la technologie Corba, et en partie dans la technologie .Net de Microsoft. C'est une donnée inhérente aux applications distribuées d'avoir du côté serveur et du côté client un substitut de l'autre. On se rappellera qu'il s'agit par ailleurs d'un des 23 design patterns mis au point par le « Gang des quatre » (présentés au chapitre 23).

Stub et skeleton

Dans chacune des technologies distribuées qui se disputent le marché, et dans leur version dite statique, le « stub » s'occupe de recevoir l'envoi, de le décomposer en les différentes parties qui le composent, puis de le transmettre. Le « skeleton », généré explicitement dans les anciennes versions de Java, ou fonctionnant implicitement dans les nouvelles, le reçoit, le recompose, l'exécute sur l'objet concerné, récupère en retour la réponse du message, qu'il transmet vers le stub. Ce dernier reprend alors la main, récupère le retour et l'intègre comme il se doit dans l'exécutable. Une autre manière de penser le stub, qui s'accorde parfaitement aux applications distribuées, lorsque celles-ci concernent des processeurs embarqués dans des machines autres qu'informatiques, est de le considérer comme un « pilote » du serveur. En effet, toute interaction de votre ordinateur avec un périphérique nécessite l'intégration d'un « pilote », qui sert d'intermédiaire à cette communication. Il reçoit les commandes du processeur, mais les ré-interprète, les temporise, les remet en forme, pour que le périphérique les « comprenne » et puisse les exécuter. Il en va également ainsi du stub dans l'interaction client-serveur. Ce stub n'est plus indispensable dans le cas de l'invocation dynamique.

Lancement du registre

Le lancement du « registre » est une opération additionnelle qui permet aux objets serveurs de s'enregistrer, et à l'application client de les retrouver. La ligne de commande `rmiregistry` s'en occupe. Sur DOS, il convient de plutôt exécuter `start rmiregistry`, de manière à pouvoir continuer de lancer des instructions sur la même fenêtre. La dernière étape côté serveur est d'exécuter le programme principal : `java PrincipaleServeurRMI`. Côté client, il faudra simplement exécuter le programme principal du client : `java PrincipaleClientRMI`. La procédure à suivre, ainsi que les différentes fenêtres DOS qui s'afficheront sur l'écran de l'ordinateur, si le tout est exécuté en local, sont montrées ci-après.

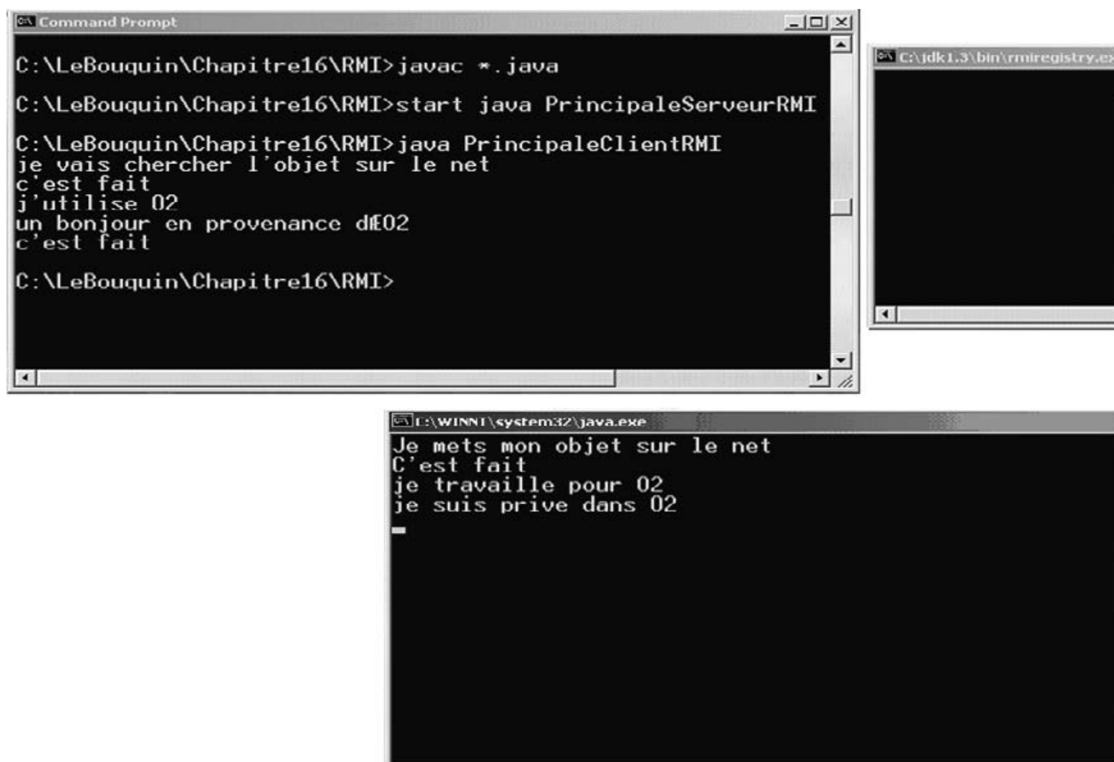


Figure 16-2

Déroulement de l'interaction RMI client-serveur, en local, et à partir d'une fenêtre DOS.

Corba (Common Object Request Broker Architecture)

Le protocole RMI de communication entre objets distribués fonctionne très bien, à ceci près que, s'il accepte en effet que des objets se parlent à travers Internet, il n'accepte, en revanche, de ne les voir se parler que dans une, et une seule, langue, le Java. Cette restriction est la condition première de la facilité d'emploi. C'est un peu court jeune homme, quand on connaît le nombre de langages informatiques qui existent aujourd'hui, et dans lesquels les informaticiens continuent à développer…

Par ailleurs, cela ne permet nullement de récupérer tout ce qui a déjà été développé, dans d'autres langages, afin de le transformer, car ce n'est finalement qu'une question d'apparence, en services disponibles sur le réseau Internet ou intranet. Corba est une réponse à ce souci d'universalité, et à ce refus de faire table rase des milliards de lignes de codes, traînant dans les sillons des disques durs. Tous les objets devraient pouvoir communiquer entre eux, qu'ils aient été écrits en Java, C++, Smalltalk, VB, Pascal, Fortran, même Cobol (ne dites pas à ma mère que je programme encore en Cobol, elle me croit programmeur Java, webmestre chez Google!). Les applications existantes, même non orientées objet, et que l'on cherche à récupérer, devraient pouvoir s'envelopper dans une sorte de revêtement objet, afin que les procédures qui les animent se métamorphosent en message.

Un standard : ça compte

Corba, non seulement, a vocation universelle mais a également vocation de standard. Au même titre qu’UML pour l’analyse et la modélisation des applications informatiques, l’OMG, toujours en quête de cet universalisme logiciel, a élu Corba standard pour les applications distribuées. L’objectif de Corba est d’assurer l’interopérabilité entre des applications hétérogènes, tant du point de vue du langage de programmation utilisé pour les rédiger que du point de vue du système d’exploitation au-dessus duquel les exécutables de ces applications tournent. Si vous voulez échanger des services à travers Internet, parlez Corba et ne parlez plus Java ou .Net (nous verrons que, si .Net de Microsoft, accepte à l’heure actuelle plusieurs langages de programmation, la présence de Windows comme OS est une condition première), du moins, si vous désirez vous faire comprendre par tout le monde.

Malheureusement, malgré l’extraordinaire avancée que constitue Corba sur le chemin de l’interopérabilité, un peu comme l’espéranto face à l’anglais, et alors que RMI et les services Web, continuent sur leur lancée, et de plus belle, Corba a un peu plus de mal à se faire entendre dans les entreprises. La raison première tient, sans nul doute, au prix à payer pour son universalité. Corba est plus compliqué que les autres protocoles à mettre en œuvre, car il exige, avant tout, d’apprendre un nouveau langage de programmation, un de plus, IDL. Et c’est, paradoxalement, cette seule complication qui nous intéresse ici, car l’acronyme IDL signifie : « Interface Definition Langage ».

Ce langage permet, en effet, d’écrire les seules lignes de code vraiment nécessaires à la réalisation d’applications distribuées : la définition des interfaces. Alors qu’en Java, vous définissez les interfaces en Java, en XML pour les services Web, en Corba, vous les définissez en IDL. Ce langage est proche du C++, et permet de complexifier davantage encore, par comparaison avec Java (mais est-ce une si bonne chose pour la diffusion de Corba ?), la manière dont vous définissez les services qu’une application serveur peut offrir à un client. Corba a trouvé une solution très classique pour se rendre indépendant de tous les langages de programmation existants : en proposer un nouveau, au-dessus de tous les autres.

Dans le cas très simple de l’exemple de ce chapitre, l’interface sera définie pratiquement comme en Java (ce code doit se trouver dans un fichier portant l’extension « idl ») :

```
module ExempleCorba {
  interface IO2Corba {
    void jeTravaillePourO2();
    string jeRenvoieUnString();
  };
};
```

IDL

Les interfaces sont déclarées au sein d’un module, ici, `ExempleCorba`. Un module constitue d’abord un espace de nommage, c’est-à-dire que tout ce qui sera déclaré à l’intérieur du module portera toujours au départ le nom du module (c’est équivalent aux « packages » ou « assemblage » en Java et aux « namespace » en C# et C++). Les modules, tout comme les répertoires, peuvent s’imbriquer les uns dans les autres. Un module peut contenir beaucoup d’autres déclarations que les interfaces, seuls éléments syntaxiques distribuables en Java.

Dans un module, IDL permet de déclarer, en plus des interfaces et des signatures de méthodes qu’ils contiennent, des constantes, de définir des types nouveaux (en récupérant le `typedef`, les `struct` ou `enum` du C++), de définir des exceptions. De plus, à la différence de Java et de C#, dans les interfaces, il est possible de rajouter des attributs. Ces attributs peuvent n’être que lisibles, « readonly », ou parfaitement modifiables par le client.

Digne héritier du C++, Corba ne force pas par sa syntaxe la pratique de l'encapsulation. Les interfaces peuvent, comme en Java et C#, être héritées entre elles, simplement ou de façon multiple. Les arguments des méthodes peuvent être passés par référence ou par valeur, ce qui peut à nouveau compliquer la traduction en Java. Lorsque le fichier `idl` est finalisé, pour passer à l'implémentation, il est nécessaire de projeter tout cela vers un langage de programmation classique. C'est là que l'universalité de Corba fait merveille : il permet en effet la « projection » du fichier `idl` dans n'importe quel langage de programmation OO ou éventuellement non OO du côté client comme du côté serveur.

En se limitant à Java et C++, un module sera traduit en package en Java et en namespace en C++. Une interface sera traduite en interface en Java mais – car on sait depuis le chapitre précédent qu'aucune structure syntaxique équivalente n'existe en C++ – en classe abstraite dans ce langage. Dans les deux langages, chaque attribut de l'interface entraînera l'existence d'une paire de méthodes pour lire et pour modifier cet attribut. Les exceptions seront traduites en exception et ainsi de suite…

Compilateur IDL vers Java

Nous allons, dans le cas d'une projection en Java, utiliser le compilateur « IDLJ » fourni par Sun (l'outil Corba offert par Sun dans le toolkit Java). Bien sûr, d'autres implémentations de Corba existent, comme Visibroker de Borland, OrbixWeb de Iona Technologies, WebSphere d'IBM, ou d'autres encore, généralement payantes. Celle de Sun est gratuite, comme le toolkit Java. Il est important de séparer les spécifications énoncées par l'OMG de l'implémentation logicielle concrète de ces spécifications, qui peut prêter à quelques variations selon les constructeurs. Nous compilons notre fichier `.idl` dans le monde Java (on pourrait le faire dans bien d'autres langages) au moyen de l'instruction suivante :

```
idlj -fall ExempleCorba.idl
```

Cette instruction a pour effet de créer un nouveau répertoire `ExempleCorba`, dans lequel plusieurs nouveaux fichiers `.class` sont installés, comme indiqué ci-après :

```
Volume in drive C is SYS
 Volume Serial Number is 0C6E-0209

 Directory of C:\Test\TestCorba\ExempleCorba

18/07/2004  16:56    <DIR>          .
18/07/2004  16:56    <DIR>          ..
18/07/2004  16:57               351 IO2Corba.java
18/07/2004  16:57             2.002 IO2CorbaHelper.java
18/07/2004  16:57               842 IO2CorbaHolder.java
18/07/2004  16:57               360 IO2CorbaOperations.java
18/07/2004  16:57             2.256 IO2CorbaPOA.java
18/07/2004  16:57             2.775 _IO2CorbaStub.java
               6 File(s)          8.586 bytes
               2 Dir(s)  21.581.393.920 bytes free
```

Ces fichiers Java doivent servir à faciliter la conception en Java de tout ce que vous avez prévu dans la définition du module. Observons-les de plus près. Tout d'abord, nous retrouvons le « stub » `_IO2CorbaStub.java qui` implémente l'interface, puisqu'elle apparaît comme un « substitut » du serveur pour le client, dans une optique tout à fait semblable à RMI. Depuis la nouvelle version de Corba, nous trouvons également un fichier `IO2CorbaPOA`. Le POA (Portable Object Adapter) sert d'intermédiaire entre l'ORB et l'objet Corba et pemet

de définir différentes « politiques » d’activation et de désactivation de l’objet serveur lors de l’exécution de ses services (un par méthode, activation uniquement lors de l’appel, problème de sollicitation concurrentielle de l’objet…). Toute application Corba doit posséder au minimum une instance de POA qui peut provenir soit directement du RootPOA (la politique par défaut) ou d’une version plus spécialisée. Associé à ce POA, un `POAManager` est à l’écoute du côté serveur. Il se conforme aux fonctionnalités du type de POA choisi pour le serveur et finalement crée l’objet serveur en le référant par un ID. Ce POA maintient une table réalisant le lien entre les ID de chaque objet et les politiques d’activation associées.

C’est le POA qui recevra la requête (il remplace donc le skeleton de l’ancienne version de Corba) et qui, en fin de compte, invoque la méthode en question en suivant sa politique d’activation et de désactivation des objets.

Le fichier `IO2CorbaOperations.java` est le plus évident à saisir, puisqu’il se limite à reproduire en Java l’interface `idl`., et ce y compris la signature des deux méthodes. Cette interface est implémentée par I`O2CorbaPOA`. `IO2CorbaHelper.java` contient un ensemble de fonctions auxiliaires, notamment la méthode `narrow()`, que nous utiliserons par la suite, et qui permet d’effectuer l’équivalent d’un « casting » dans le type de l’interface. Finalement, le fichier `IO2CorbaHolder.java` a comme raison d’être de traiter les arguments des méthodes qui peuvent être passés par référent, et qui ne sont pas pris en compte de manière automatique par Java.

Il reste maintenant à concrétiser le côté client et le côté serveur de l’application, en écrivant deux nouveaux fichiers : `ExempleCorbaClient.java` et `ExempleCorbaServeur.java`, que nous allons détailler avec plus d’attention, car il s’agit bien d’un travail que le programmeur est d’office amené à faire.

Côté client

Le fichier ExempleCorbaClient.java

```
import ExempleCorba.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ExempleCorbaClient {
  public static void main(String args[]) {
    try {
      ORB orb = ORB.init(args, null); /* creation d'un objet CORBA */
    /* ensuite débute une suite d'instructions pour la mise en oeuvre du nommage */
      org.omg.CORBA.Object objRef =  orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
      NameComponent nc = new NameComponent("unObjetO2", "");
      NameComponent path[] = {nc};
    /* il faut retrouver l'objet sur lequel déclencher les services */
      IO2Corba unObjetO2 = IO2CorbaHelper.narrow(ncRef.resolve(path));
      System.out.println("j'utilise O2");
    /* on envoie les deux messages */
      unObjetO2.jeTravaillePourO2();
      System.out.println(unObjetO2.jeRenvoieUnString());
    }
    catch(Exception e) {
      System.out.println("Error : " + e);
      e.printStackTrace(System.out);
    }
  }
}
```

Quelques « imports » sont d’abord nécessaires pour récupérer les fonctionnalités Corba, par exemple, le service de « nommage » prévu dans `org.omg.CosNaming.*`. Nous plaçons l’essentiel du code dans un bloc `try-catch` de gestion d’exception, puisque de nombreux problèmes pourraient survenir dans la communication entre le client et le serveur. Il faut ensuite créer un objet Corba, `orb`, pour pouvoir utiliser le bus Corba, qu’on appelle l’ORB (l’Object Request Broker), et qui permet l’empaquetage et le désempaquetage des messages, et la circulation de ceux-ci sur TCP/IP. Le service le plus important que Corba met à notre disposition est le service de « nommage » qui permet, comme dans le cas du registre RMI, de récupérer un objet par ses nom et adresse, où qu’il se situe sur Internet.

Les services de Corba

Corba, répondant ainsi aux spécifications de l’OMG, met d’autres services à notre disposition, tous implémentés en partie par les constructeurs, tels que le service de nommage, mais aussi le « cycle de vie », qui définit la manière dont les objets sont créés, déplacés ou copiés, le « service d’événements », qui permet à des objets de répondre à des événements, le « service de transaction », qui permet à une succession de messages de s’inverser si une étape se passe mal, le « service d’accès concurrentiel », qui permet à un même objet de traiter simultanément plusieurs clients, un « service de requête », par lequel l’objet peut nous informer sur son état et ses méthodes, et d’autres encore.

Nous n’allons pas passer trop de temps sur la manière dont les objets utilisent ce service de nommage. Il faut d’abord récupérer un « naming context », car les noms des objets peuvent se structurer différemment selon l’implémentation spécifique de Corba qu’on utilise. Un tableau de `NameComponent` reprendra le nom entier de l’objet. Ici, nous nous limiterons à juste un nom : `unObjet02`, le même nom d’objet que nous avions donné lors de l’utilisation de RMI. Notre tableau se réduit ici à un seul élément.

L’équivalent du `unObjet02 = (IO2RMI)Naming.lookup("unObjet02")` de RMI se transforme ici en `IO2Corba unObjet02 = IO2CorbaHelper.narrow(ncRef.resolve(path))`. L’effet est le même : réaliser l’association entre un objet local dont le référent est `unObjet02` et l’objet distant, lequel recevra *in fine* les messages. C’est cette pratique qui permet de rendre l’écriture d’une application distribuée très proche de l’écriture d’une application s’exécutant en local. La classe `IO2CorbaHelper` permet de réaliser le « casting » de l’objet dans l’interface désirée.

Finalement, l’invocation des deux méthodes se fait exactement comme pour RMI, sans différence aucune avec une application locale. Cette écriture simple dissimule pourtant une procédure laborieuse, comprenant le codage des messages et des arguments dans une forme transférable sur le réseau, le transport de ces messages à travers le Web, l’activation et l’exécution de ces méthodes sur l’objet serveur, la récupération des « retours », le codage de ceux-ci, leur retransmission sur le Web vers le client, et finalement la récupération de ces « retours » par le client.

Côté serveur

Le fichier ExempleCorbaServer.java

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import ExempleCorba.*;
```

```
public class ExempleCorbaServer{
  public static void main(String args[]){
    try {
      ORB orb = ORB.init(args, null);   /* creation d'un objet CORBA */
       /* creation de l'objet serveur */
      O2CorbaServant o2Ref = new O2CorbaServant();
       /* Obtention d'une référence au rootpoa et activation du POA manager */
      POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(o2Ref);
      IO2Corba href = IO2CorbaHelper.narrow(ref);
       /* Obtention d'une référence à l'objet serveur */
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
      NameComponent path[] = ncRef.to_name("unObjetO2"); // dénomination de l'objet serveur

      System.out.println("Je mets mon objet sur le net");
      ncRef.rebind(path, href);       /* on enregistre l'objet serveur */

      orb.run();    // on attend l'invocation des clients
      System.out.println("c'est fait");
    }
    catch(Exception e) {
      System.err.println("Error: " + e);
      e.printStackTrace(System.out);
    }
  }
}

class O2CorbaServant extends IO2CorbaPOA { /* implémentation de l'objet serveur */
  public void jeTravaillePourO2() {     /* implémentation du premier service */
    System.out.println("je travaille pour O2");
    jImplementeLeServiceO2();
  }
  public String jeRenvoieUnString() {   /* implémentation du deuxième service */
    return "un bonjour en provenance d'O2";
  }
  private void jImplementeLeServiceO2() {
    System.out.println("je suis prive dans O2");
  }
}
```

Comme pour le fichier du côté client, nous devons, dans un premier temps, créer un objet Corba, un `orb`. Une différence importante avec le protocole RMI est que Corba sépare les responsabilités du côté du serveur en un ensemble d'objets « servant » et un « serveur », à proprement parler, qui a pour rôle d'instancier ces objets servants selon le protocole d'activation du POA. Ce sont les objets servants qui implémentent l'interface `IO2CorbaPOA` responsable de la définition des services à délivrer.

Deux classes sont donc à l'œuvre ici, la classe `serveur` qui s'occupe de créer les objets servants pour les rendre disponibles sur le bus Corba, et la classe `servant` qui type les objets rendant les services. C'est par l'instruction `org.omg.Corba.Object ref = rootpoa.servant_to_reference(o2Ref);` que le `servant` se rend disponible

sur le serveur doté du POA choisi. Le service de nommage fonctionne comme pour le client. L'enregistrement de l'objet `servant`, avec le nom symbolique qui permettra de le référer sur Internet, se fait par l'instruction : `ncRef.rebind(path, href);` équivalente à l'instruction `Naming.rebind("unObjet02", this);` du protocole RMI (on retrouve l'expression `rebind`). À ce stade, la dernière instruction est de lancer le serveur et d'attendre simplement qu'un client adresse une requête. C'est de cela dont s'occupe l'instruction :

```
orb.run();.
```

Exécutons l'application Corba

De manière à faire tourner cette application, il faut d'abord compiler tous les fichiers Java, ceux du répertoire `ExempleCorba`, et les deux que l'on vient de réaliser, installés quant à eux, dans le répertoire supérieur. Au même titre que le `rmiregistry` de RMI, il faut déclencher le service de nommage par la ligne de commande suivante : `tnameserv`. Ensuite, les deux dernières étapes sont, toujours comme en RMI, l'exécution du serveur suivie de l'exécution du client. Si tout se déroule comme attendu, les commandes à écrire, ainsi que les résultats obtenus dans un environnement DOS, devraient apparaître comme figure 16-3.

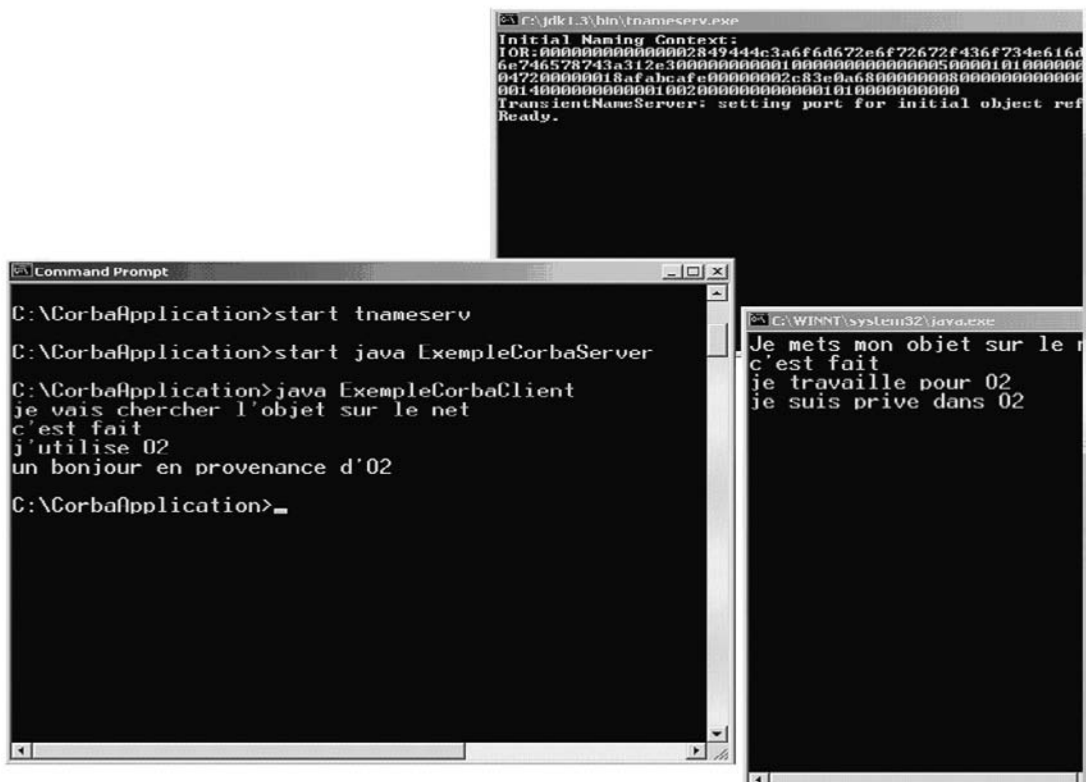


Figure 16-3

Déroulement de l'interaction Corb client-serveur, en local, et à partir d'une fenêtre DOS.

Corba vise à offrir un environnement d'exécution pour des millions d'objets à granularité variable : de simples instances d'objet C++, de quelques centaines de bits, jusqu'à des objets plus volumineux, encapsulant des millions de lignes de code Cobol, déjà programmées et à récupérer par un emballage IDL. Bien entendu, tous ces objets ne sont pas utilisés en même temps. Ainsi, pour éviter d'encombrer inutilement la mémoire centrale, Corba propose également plusieurs stratégies d'activation des objets définies par le POA.

Par défaut, c'est lors de l'envoi de message sur le serveur que Corba activera l'objet pour le rendre capable de traiter le message. Mais plusieurs stratégies d'activation restent possibles, à choisir par le programmeur quand il développe la partie serveur, comme un processus par activation d'objet, un processus partagé par plusieurs activations, ou, encore, un processus par exécution de méthodes sur un objet.

Ce même exemple est bien évidemment transposable dans n'importe quel langage de programmation OO, que cela soit côté serveur ou côté client.

Corba n'est pas polymorphique

Un autre point particulièrement intéressant ici, car il touche aux fondements de l'OO, est la façon dont RMI, au contraire de Corba, permet un vrai polymorphisme, grâce à la possibilité de transférer entre le client et le serveur, non plus, seulement, des signatures de méthodes empaquetées, mais, par exemple, tout le code d'une classe. Pourquoi est-ce nécessaire à l'implémentation du polymorphisme ? Supposez qu'en réponse à un message envoyé par le client, le serveur renvoie un objet typé dynamiquement (c'est-à-dire pendant l'exécution), `FilsO1`. Or, dans la signature du message, déclarée dans l'interface, le type statique de ce retour est la superclasse `O1`, qui est la seule connue par le client.

Recevant cet objet, vers lequel un nouveau message propre à la classe `FilsO1` (c'est-à-dire redéfini dans la classe `FilsO1`) pourrait être envoyé, il est nécessaire de connaître le code de ce message. La classe `FilsO1` n'étant pas connue par le client, elle sera téléchargée pendant l'exécution du programme. RMI permet donc, dans le feu de l'action, de compenser par un transfert de code ce qui manque comme information, soit chez le serveur, soit chez le client.

Le polymorphisme en est directement responsable, car il permet de spécifier, seulement pendant l'exécution, parmi plusieurs méthodes celle qui devra réellement être exécutée. Corba ne le peut pas, car les codes des classes ne peuvent être transférés côté client ou côté serveur en cours d'exécution. C'est la raison pour laquelle RMI, là encore à la différence de Corba, génère les « stubs » et les « skeletons » à partir des classes implémentant les interfaces, et non plus directement à partir des interfaces. La possibilité pour un code Java distribué de créer ce « stub », sur le vif, face à une classe manquante du côté serveur ou client, et de le transférer vers l'un ou l'autre, est la base du fonctionnement de RMI et de Jini, comme nous allons le voir.

Rajoutons un peu de flexibilité à tout cela

Nous avons établi dans le chapitre précédent une analogie entre le rôle du « stub » et celui d'un pilote de périphérique. Essayons de la pousser un peu. Prenons, par exemple, le cas bien connu de l'imprimante. Avant de pouvoir utiliser l'imprimante d'un ordinateur, il faut charger le pilote, qui est un programme servant d'intermédiaire entre l'imprimante et le processeur.

Reprenons notre exemple de l'appareil photo et de l'imprimante, de plus en plus probable, avec l'accroissement de la mobilité de tous les utilitaires embarquant un processeur. Vous arrivez dans un lieu quelconque avec votre portable, mais cela pourrait être votre agenda électronique, votre appareil photo, et vous désirez, sans plus attendre, imprimer un document, contenu dans l'un de ces appareils. Dans une ville, il est possible

que plusieurs imprimantes soient disponibles dans un rayon de 100 m, presque à portée de main, toutes pouvant imprimer votre document. Comment choisir ? Plusieurs critères pourraient dicter votre choix : la proximité, la qualité de l’impression, le prix, et, si le document est volumineux, la durée d’impression.

L’idéal serait que sur votre portable ou dans le viseur de votre appareil photo, à la suite d’une petite investigation de votre part sur des possibles « services » d’impression existant à proximité, vous voyiez apparaître une liste des imprimantes possibles, chacune avec son emplacement, son prix, sa qualité et sa durée d’impression. Une fois votre choix effectué (vous pourriez même imaginer que votre portable ou votre appareil photo connaît vos préférences au point de pouvoir automatiser ce choix), c’est seulement à ce moment-là, qu’il devient nécessaire de télécharger le pilote de l’imprimante, le temps de l’interaction de votre portable ou de l’appareil photo avec celle-ci.

Il est crucial que vous soyez également tenu informé des soudaines défaillances d’un appareil, jusqu’alors disponible sur le réseau, que vous le soyez aussi du rajout dans le réseau, d’une nouvelle imprimante, ou de la disparition d’une d’elles du marché. Bref, il faudrait avoir la possibilité de ne charger un « stub » ou un « pilote » qu’à l’issue d’un choix entre plusieurs d’entre eux, de ne le conserver que le temps de l’exécution, et de pouvoir à tout moment remettre ce choix en question.

Tout cela conduit à une vision de l’informatique plus mobile, souple, adaptable et, surtout, plus en phase avec les innovations technologiques de type réseau sans fil, la mobilité des utilisateurs et l’accélération des changements que la pression économique induit. Les applications informatiques évoluent, en effet, très vite, et il est important d’être tenu au courant des nouveautés dont vous pourriez tirer un profit immédiat. Il faudrait pour cela pouvoir rapidement charger un nouveau stub, sans vouloir le figer à jamais sur votre disque dur, et sans qu’il entre en compétition avec ceux qui le précédaient (suivez le regard vers certaines .dll que nous ne nommerons pas).

Que les applications informatiques soient plutôt à louer, le temps de leur utilisation, qu’acquises définitivement est en passe d’entrer dans les mœurs de nombreuses entreprises, lassées de ces mises à jour incessantes et coûteuses de suites bureautiques ou autres logiciels graphiques. Cette nouvelle vision ne s’accorde pas avec la pratique trop statique, consistant à télécharger le stub avant l’exécution du programme, et à partir d’une interface déjà identifiée.

Corba : invocation dynamique versus invocation statique

Corba, le premier, a infléchi cette contrainte, en permettant que le stub soit comme généré pendant l’exécution du client, et ne soit plus un préalable à cette exécution. On parle alors d’invocation dynamique en lieu et place de l’invocation statique que nous avons illustrée dans notre petite application précédente. Le client Corba peut, à partir d’objets distribués dont il connaît l’existence et l’adresse, spécifier le nom de la méthode devant être invoquée, ainsi que les paramètres désirés.

Pour peu qu’un tel objet existe, un stub et un skeleton seront générés tant du côté client que serveur, permettant l’envoi de messages, exactement comme dans le cas statique. Toutes ces informations sont stockées dans une sorte d’annuaire des services (l’interface repository) associés à chaque objet entre lesquels le client pourra faire son choix. Ainsi le client peut-il toujours, pendant l’exécution, se renseigner sur ce qui est disponible, construire son message, et l’envoyer vers le serveur de son choix. Dans le cas d’une invocation dynamique, dont la réalisation est nettement plus complexe que celle de l’invocation statique (et dépasse le cadre de cet ouvrage), pendant son exécution, le code client peut obtenir la référence d’un objet Corba, instancier une variable de type `Request`, obtenir l’interface de l’objet à partir de « l’interface repository », en extraire les méthodes et les attributs et, à partir de ceux-ci, construire la requête, l’invoquer et finalement obtenir les résultats de son exécution en retour. Le tout sans aucune compilation préalable pour adapter le client au serveur.

Jini

Du côté de Java-RMI, Sun a également accru la souplesse de l’approche, avec la possibilité de découvrir des services plutôt que de s’y conformer dès le départ, comme l’exige le téléchargement du stub en préambule de l’interaction. En quelques lignes, car il en faudrait tellement plus pour rendre justice à cette avancée technologique importante, Jini doit tout d’abord être perçu comme la continuation de RMI. Jini comprend une addition structurelle, essentielle à son fonctionnement, comme en Corba un annuaire de services (appelé « lookup »).

Ce dernier simplifie la médiation entre le client et le serveur, en gardant continûment la liste des services disponibles sur le réseau, et en permettant au client de s’informer pour choisir le service qui réponde le mieux à ses attentes. Rappelez-vous la petite histoire de l’imprimante. Seule une approche de type Jini permet à ce scénario de se réaliser. Cet « annuaire » doit également prendre en charge toutes les modifications se produisant sur le réseau de service, et en informer les clients le souhaitant : disparition, addition ou modification des services existants.

L’utilisation de Jini se déroule de la manière suivante. D’abord, si un artefact quelconque cherche à rendre ses services disponibles sur un réseau, il doit découvrir un annuaire pour y enregistrer son offre. Soit il possède l’adresse IP de cet annuaire, soit il se met en quête dans le réseau local d’un annuaire disponible. Une fois cet annuaire découvert, l’artefact met en dépôt sur celui-ci un « proxy », qui peut être soit un stub, permettant l’interaction directe entre un client et lui, soit, plus simplement encore, l’ensemble du service exécutable, c’est-à-dire, un objet et les méthodes à exécuter sur celui-ci.

En possession de cet ensemble, le client pourra exécuter le service, sans plus avoir à passer par le serveur. Lorsque, à son tour, un client (rappelons-nous que les rôles, comme dans la vision peer-to-peer, sont tout à fait interchangeables) désire s’offrir un de ces services, il passe également par ce même annuaire, afin de s’informer sur l’offre disponible. Il transmet à l’annuaire les services désirés, de manière à récupérer le proxy qui lui permettra de les utiliser. Ce proxy sera, soit le service complet, soit le stub qui, par RMI, servira de pont entre le client et le serveur.

Un problème important à solutionner devient alors la robustesse de cette architecture flexible, car les services peuvent apparaître ou disparaître à tout moment. Il faut imaginer un mécanisme qui tolère cette flexibilité, tout en limitant les impacts nuisibles qu’elle peut causer. Lors d’un envoi de message qui échoue, Jini, dans le prolongement de Java, délègue à un mécanisme de gestion d’exception la possibilité de remédier à cet échec. Par ailleurs, un service qui décide de quitter le réseau en avisera l’annuaire qui détient son proxy, de manière que celui-ci soit mis hors d’état. L’annuaire avisera les utilisateurs courants de ce proxy qu’il est temps de mettre fin à cette utilisation.

Un service pourrait également quitter le réseau, de manière plus brutale, sans en informer l’annuaire. Les dommages causés par un tel départ sont atténués par un mécanisme dit de « leasing », qui demande à l’artefact qui fournit le service d’informer l’annuaire sur la durée de mise à disposition du service, et de la fréquence selon laquelle l’artefact se rappellera au bon souvenir de l’annuaire. Si lors d’une de ces sessions de contact, le service ne peut plus être contacté ou s’il ne désire pas renouveler le bail de son proxy, ce dernier sera détruit, et les clients seront avertis de sa disparition.

XML : pour une dénomination universelle des services

Un dernier problème à relever concerne la façon dont les services sont présentés sur l'annuaire, et celui dont le message ainsi que son retour sont codés. Corba le fait à sa sauce, Jini à la sauce Java, mais rien de tout cela n'est vraiment conforme au style dans lequel les concepteurs et les acteurs principaux du Web ont décidé de coder toute information installée et circulant sur ce Web. Pour autant que les services à disposition, ainsi que les messages qui circulent sur Internet, soient à assimiler à n'importe quel autre type d'information disponible sur le Web, un standard aujourd'hui s'impose, qui a pour nom XML. La circulation ne se fait plus véritablement via Internet mais via le Web, en utilisant le protocole http. Dans l'approche des services Web, tout comme dans n'importe quel échange au-dessus du protocole http, un « serveur Web » est indispensable pour recevoir et traiter la requête côté serveur (par exemple : « Apache » sur Linux ou « IIS » sur Windows »). Alors que Corba fait jouer à l'ORB le rôle de bus de communication des requêtes entre clients et serveurs, dans le cas des services Web, ce rôle incombe au protocole http, à travers lequel ne circule que du texte et non plus des ordres prêts pour l'exécution, comme c'est le cas pour Corba ou RMI.

Par un jeu de balises imbriquées (voir encart), XML permet de structurer de manière très homogène tout le contenu sémantique (à différencier de sa seule mise en forme) des documents disponibles sur le Web. XML pourrait ainsi constituer un mode de représentation des services et des messages envoyés sur le Web, à utiliser au-dessus des modes de représentation propres à Corba ou RMI. Cela permet également d'homogénéiser, dans leur représentation, tous les services disponibles sur le Web, quelle que soit l'implémentation finale de ces services : Corba, RMI ou DCOM. L'interopérabilité redevient possible à partir d'XML. Ce mariage entre l'informatique distribuée, la possibilité qu'ont deux applications informatiques de se solliciter mutuellement à travers le Web et le langage XML porte le nom de « services Web ».

Cela permet aussi d'assimiler tout service à n'importe quel type de documentation circulant à travers le Web, et de bénéficier ainsi des mécanismes de recherche de documentation et d'extraction d'information (par un « parsing » des documents XML), devenus aussi indispensables que courants sur le Web.

Microsoft a parfaitement anticipé avec XML cette homogénéisation du contenu des messages et des services dans sa nouvelle plate-forme .Net, en automatisant la traduction en XML de tous services codés, au départ, dans un langage de programmation comme C# ou VB.Net. Sun fait de même, en proposant un ensemble de librairies Java dénommées JAX-RPC (concaténant Java, XML et RPC – Remote Procedure Call), et permettant également une parfaite interopérabilité entre les langages de programmation et les systèmes d'exploitation. PHP ne pouvait pas ne pas suivre étant donnée l'importance du monde Web pour ce langage, et les services Web ainsi que les protocoles (Soap…) qui les accompagnent sont parfaitement intégrés.

Nous nous limiterons dans la suite à décrire la manière dont .Net nous invite à développer des services Web, la plus simple à mettre en œuvre, en prenant conscience que cette manière d'intégrer XML dans la dénomination des services et des messages est en passe d'être adoptée par tous les grands constructeurs informatiques, tous les langages de programmation, et de devenir *de facto* un standard Web.

Tim Berners-Lee et le Web sémantique

Dans cette dernière partie de chapitre, nous avons fait allusion à un souci permanent accompagnant le développement explosif du Web. Il s'agit de l'uniformisation de la structure des documents qui y sont accessibles, y compris des documents reprenant la signature d'objets exécutables mis à notre disposition sur certaines machines. Voir le Web comme un dépositaire d'une infinité d'informations, qui ne se limitent pas à apparaître (comme de simples photos dans un album), mais qui, de surcroît, facilitent leur exploitation par des humains dans le plus de contextes opérationnels possibles, cela a toujours été la préoccupation essentielle de Tim Berners-Lee, son inventeur. Ce dernier veut rendre le Web le plus utile qui soit aux humains, en rendant toute l'information qu'il contient facilement accessible et surtout traitable par les programmes informatiques eux-mêmes. Il chérit son enfant comme tout bon père le fait, en espérant, qu'en grandissant, il prenne la direction souhaitée, même si, toute éducation ne fait qu'esquisser quelques chemins de vie possibles parmi lesquels, non seulement l'enfant effectue son choix, mais dévie très vite des premières balises rencontrées. En matière d'autonomie et de contrôle difficiles, le Web est incontestablement un sommet.

Tim Berners-Lee, diplômé de l'université d'Oxford, a consacré les premières années de sa vie professionnelle à l'élaboration de solutions de type réseau temps réel, technologie code-barres, traitement de texte, système d'exploitation multitâche, et bien d'autres secteurs de l'informatique. C'est en 1980, lors d'un séjour de six mois comme consultant informatique du CERN (le laboratoire européen de physique des particules), qu'il écrit un programme appelé « Enquire », qui permet à des documents, non seulement d'encoder leur organisation spatiale et leur visualisation, mais également, basé sur la technologie hypertexte, de se référencer mutuellement. Le protocole http avait vu le jour. Il faudra cependant attendre encore dix ans pour que Tim Berners-Lee publie officiellement cette idée et l'impose comme la principale utilisation d'Internet et sans doute la plus aboutie (à côté des protocoles ftp, e-mail, telnet…).

En 1994, il fonde le consortium W3 qui préside aux destinées d'http et veille à ce que le Web évolue comme le souhaite son créateur. Il en assure donc la direction depuis sa création. Il est également détenteur d'une chaire de recherche au MIT. La récriture des signatures des messages sous forme d'XML s'inscrit parfaitement dans cette préoccupation d'uniformisation de tout document circulant sur le Web. Écrire les exécutables de la même manière que tout autre document permet d'étendre tous les systèmes de recherche et d'extraction des documents à ces mêmes exécutables. Le futur des agents, prétendument intelligents, et circulant sur le Web au service de l'un ou l'autre utilisateur, ne pourra s'en trouver que facilité. C'est la vision ultime du « Web sémantique », que toutes les données qui s'y trouvent puissent être définies et reliées d'une manière qui les rende facilement utilisables par les applications logicielles. Une première uniformisation sera de type syntaxique et anticipée par des langages comme XML, qui permettent de simplifier la structuration des documents et de les détacher de leur visualisation. Une seconde, extraordinairement plus ambitieuse, se doit d'être de type sémantique, et viser à la terminologie et au vocabulaire employé, terminologie unique, dont il faudra forcer l'adoption par de multiples communautés d'utilisateurs qui, aujourd'hui, se comprennent plus entre les lignes que grâce aux lignes. Cette terminologie, que l'on tente d'organiser en ontologies reprenant les termes d'un domaine et les relations logiques entre ces termes, devra faciliter tant l'indexation que la recherche des services Web, et résoudre le problème que rencontrent aujourd'hui les moteurs de recherche de sites Web. Cette simplification se trouve anticipée, encore très timidement, par des solutions comme les DTD ou les schémas XML et la mise au point du RDF (Resource Description FrameWork, sorte de résurrection des réseaux sémantiques inventés par l'IA dans les années 1970). L'auteur, plébiscité par le magazine *Time*, comme l'un des 100 plus grands esprits de ce siècle, défend cette vision dans un récent ouvrage intitulé *Weaving the Web* et publié chez Hardcover. Depuis fin 2004, il détient une chaire de professeur au département d'électronique et d'informatique de l'université de Southampton et y poursuit son projet d'enrichissement sémantique du Web. Il est aussi un ardent défenseur d'un Web préservant une parfaite « neutralité » quant aux équipements qui s'y connectent et au contenu de l'information qui le traverse, un Web libertaire, démocratique et gratuit ! 2004 fut une très bonne année pour lui car il s'est vu également annobli par « *Ze British Queen herself* » qui enfin et grâce à lui, peut utiliser Facebook.

XML

XML installe l’information dans une structure imbriquée de balises, comme l’exemple ci-après l’illustre, lorsqu’il s’agit d’encoder un livre et ses auteurs :

```
<livre>
  <auteur>
    <prénom> Hugues </prénom>
    <nom> Bersini </nom>
  </auteur>
  <auteur>
    <prénom> Ivan </prénom>
    <nom> Wellesz</nom>
  </auteur>
</livre>
```

Il est très facile dans une telle structure récursive d’effectuer une recherche ou de traiter l’information. L’utilisation massive d’XML devrait permettre une bien plus importante homogénéisation de toute l’information contenue dans Internet, impossible par HTML (qui est une simple manière d’organiser la disposition de cette information mais non pas de définir son contenu). La dénomination des balises est laissée au libre choix des concepteurs. Cependant, l’intérêt est de s’accorder sur des dénominations et des structurations de document communes, qui seront définies dans un format appelé DTD (Data Type Dictionnary), par exemple, toujours pour le livre :

```
< !ELEMENT livre (auteur)+>
< !ELEMENT auteur(prénom, nom)>
< !ELEMENT prénom (#PCDATA)>
< !ELEMENT nom (#PCDATA)>
```

Une autre possibilité, plus récente, d’écriture des documents de conformation a été adoptée dans le cadre des services Web. Ce sont les schémas XML, plus proches dans leur syntaxe des documents XML de base (à nouveau, un système de balises imbriquées, ce qui permettra d’uniformiser le traitement). En permettant la prise en compte de type de données plus complexes que la simple composition récursive propre à XML, ces schémas faciliteront la traduction dans un format XML des bases de données relationnelles ainsi que des diagrammes de classe UML. Le schéma XML de l’exemple du haut ressemblerait plus ou moins à ceci :

```
<Schema xmlns = « urn :schemas-microsoft-com :xml-data »>
 <ElementType name= “prénom” content = “textOnly” model = “closed” />
 <ElementType name= “nom” content =”textOnly” model = “closed” />
 <ElementType name = “auteur” content = “eltOnly” model = “closed”>
    <element type = “nom” minOccurs = “1” maxOccurs = “1” / >
    <element type =  “prenom” minOccurs = “1” maxOccurs = “1” / >
  </ElementType>
  < ElementType name= “livre” content = “eltOnly” model = “closed” >
    <element type = “auteur” minOccurs = “0” maxOccurs = “*” />
  </ElementType>
</Schema>
```

Les services Web sur .Net

La plate-forme .Net de Microsoft a pour vocation de fournir un environnement qui simplifie la conception, le développement, le déploiement et l'exécution d'applications distribuées. Corba, RMI et Jini ont en commun de concevoir ces applications distribuées en termes d'invocation de méthodes à distance. Ces appels de méthodes et envoi de messages seront perçus – dans la vision Microsoft, partagée par HP, IBM, Sun, PHP, et plusieurs autres acteurs mammouths du monde informatique – comme la mise à disposition de services Web. La nouveauté essentielle par rapport à Corba ou Jini est l'exploitation intensive d'XML comme langage de description de ces services.

Reprenons notre exemple de RMI et de Corba et développons en C#, cette fois, un service que nous désirons rendre disponible sur le Web.

Code C# du service

Fichier TestService.asmx

```
<%@ WebService Language="C#" Class="TestService" %>

using System;
using System.Threading;
using System.Web.Services;

public class TestService : WebService {

          [WebMethod]
          public string jeTravaillePourLeWeb (String unNom)
          {
                  return "Salut, " + unNom + jeSuisPriveDansLaClasse();
          }

          private string jeSuisPriveDansLaClasse() {
                  return ", je travaille pour le web";
          }
}
```

Un service Web doit se coder dans un fichier de type `asmx`. À la différence de Corba et de RMI, qui exigent de débuter l'écriture des services sur le Web par la définition d'une interface, suivie pour son implémentation d'une classe donnée dans laquelle est défini le corps d'exécution, .Net permet de partir directement de l'implémentation. La présence de `[WebMethod]` rend la signature de la méthode disponible sur le Web, sans qu'il y ait besoin de détacher cette signature pour l'installer dans un code à part (on dit que l'on « expose » la méthode sur le Web).

De manière à rendre ce service disponible, mais surtout « lisible », sur le Web, .Net crée automatiquement une version XML de ce même service, reprenant ce qu'il y a lieu de connaître pour utiliser ce service : son nom, les arguments à passer et ce que le service renvoie en retour. Le type de langage XML utilisé à cette fin s'appelle Web Service Description Language: WDSL. Ci-après, vous pouvez voir une partie du fichier `TestService.asmx?WSDL` qui reprend la description du service.

```
<?xml version="1.0" encoding="utf-8" ?>
- <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap="http://
schemas.xmlsoap.org/wsdl/soap/" xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://
tempuri.org/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tm="http://
microsoft.com/wsdl/mime/textMatching/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
targetNamespace="http://tempuri.org/" xmlns="http://schemas.xmlsoap.org/wsdl/">
- <types>
- <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
- <s:element name="jeTravaillePourLeWeb">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="unNom" type="s:string" />
  </s:sequence>
  </s:complexType>
  </s:element>
- <s:element name="jeTravaillePourLeWebResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="jeTravaillePourLeWebResult" type="s:string" />
  </s:sequence>
  </s:complexType>
  </s:element>
  <s:element name="string" nillable="true" type="s:string" />
  </s:schema>
  </types>
- <message name="jeTravaillePourLeWebSoapIn">
  <part name="parameters" element="s0:jeTravaillePourLeWeb" />
  </message>
- <message name="jeTravaillePourLeWebSoapOut">
  <part name="parameters" element="s0:jeTravaillePourLeWebResponse" />
  </message>
- <service name="TestService">
- <port name="TestServiceSoap" binding="s0:TestServiceSoap">
  <soap:address location="http://localhost/6152/TestService.asmx" />
  </port>
  </definitions>
```

WDSL

Parmi les différentes informations encodées en XML (nous n'en détaillerons pas la structure) vous pouvez deviner le nombre et le type des arguments, le retour du service ainsi qu'à la fin, l'emplacement URL de ce dernier. Ici, car nous travaillons en local, cet emplacement est : *`http://localhost/6152/TestService.asmx`*.

Afin de visualiser les services en format WDSL, il faut créer une URL virtuelle, ici le `localhost/6152` et éditer le fichier `TestService.asmx?WSDL` à partir de votre navigateur. La lecture du service Web sur le navigateur n'est possible que si le « serveur web » est actif (Apache sous Linux ou IIS sous Windows). En effet, c'est lui qui reçoit la requête, l'interprète comme un « service web » et vous en expose le contenu. Mais, bien évidemment, l'adresse URL de ce service variera en fonction de l'emplacement de l'objet à même de l'exécuter. Au même titre que Corba ou RMI, le service doit pouvoir être localisé sur Internet, mais cette localisation est directement codée sous forme XML et fait partie intégrante de la définition du service. Nous justifierons la présence de l'expression Soap par la suite.

> **WDSL**
> L'existence de ce standard de description de services, WDSL, rendra toute application distribuée utilisable par l'ensemble des technologies d'objets distribués, tout environnement Web (par exemple, ce service pourrait être utilisé à partir d'un navigateur Internet) et sur toute plate-forme informatique confondue.

Création du proxy

Une fois le service disponible sur Internet et prêt à être exécuté sur un serveur donné, comment un client peut-il y avoir accès ? Comme pour RMI et Corba, il faut créer un « proxy » ou un « stub », qui permettra au client, localement, de procéder, comme s'il s'adressait directement au serveur. C'est ce proxy qui sert de passerelle entre le client et le serveur. Comme dans tous les mécanismes d'invocation statique d'objet distribué décrits jusqu'à présent, c'est un intermédiaire essentiel. Ce proxy, créé par « rmic » en RMI et par « idlj » en Corba, se construit dans .Net de la manière suivante :

```
C:\TestService>wsdl /l:cs /o:TestServiceProxy.cs http://localhost/6152/TestService.asmx?WDSL

Microsoft (R) Web Services Description Language Utility
[Microsoft (R) .NET Framework, Version 1.0.3705.0]
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Writing file 'TestServiceProxy.cs'.

C:\TestService>
```

Le proxy se crée à partir de l'instruction `wsdl` et s'installe dans un fichier `TestServiceProxy.cs`. Remarquez la localisation Internet du fichier asmx.

Toute classe devant être utilisée par une autre dans Windows se doit d'être transformée en une .dll. Il faut donc, du côté client maintenant, compiler ce proxy et le transformer en une .dll, pour qu'il puisse être utilisé par le client. Cela se fait au moyen de l'instruction suivante :

```
C:\TestService>csc /out:TestServiceProxy.dll /t:library /r:system.web.services.dll TestServiceProxy.cs

Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

C:\TestService>
```

Le proxy, côté client, est maintenant prêt à jouer son rôle d'intermédiaire entre le client et le serveur. Il reste encore à créer le code du client, comme indiqué ci-après :

Code C# du client

TestClient.cs

```
using System;
public class TestClient {
  public static void Main() {
    TestService unTest = new TestService();
    Console.WriteLine(unTest.jeTravaillePourLeWeb(" moi le service "));
  }
}
```

Il faut compiler ce code en le rattachant au proxy, comme indiqué ci-après :

```
C:\TestService>csc /r:TestServiceProxy.dll TestClient.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

C:\TestService>
```

Il ne nous reste plus qu'à exécuter le client :

```
C:\TestService>TestClient
Salut, moi le service, je travaille pour le Web

C:\TestService>
```

Et le tour est joué.

Soap (Simple Object Access Protocol)

Soap est le protocole XML d'écriture des messages à envoyer au serveur (le nom des méthodes et leurs paramètres) et d'écriture de la réponse obtenue, suite à l'exécution des messages. Les deux paquets Soap d'appel et de réponse de la méthode sont reproduits ci-après :

```
POST /6152/TestService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/jeTravaillePourLeWeb"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3
➥.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <jeTravaillePourLeWeb xmlns="http://tempuri.org/">
      <unNom> moi le service </unNom>
    </jeTravaillePourLeWeb>
  </soap:Body>
</soap:Envelope>
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/
2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <jeTravaillePourLeWebResponse xmlns="http://tempuri.org/">
      <jeTravaillePourLeWebResult> Salut, moi le service, je travaille pour le Web </jeTravaillePourLeWebResult>
    </jeTravaillePourLeWebResponse>
  </soap:Body>
</soap:Envelope>
```

Ce message Soap sera transmis au proxy qui le traduira comme il se doit pour le transmettre au serveur. On conçoit que l'utilisation de « parseur » XML soit indispensable, de manière à extraire les informations nécessaires pour transmettre le message au serveur, son nom et ses arguments. Une fois ce message exécuté et la réponse obtenue, cette dernière sera empaquetée à son tour dans un format Soap, que le proxy dépaquettera, afin de la rendre disponible dans le code client.

Invocation dynamique sous .Net

Malgré l'existence du proxy et des compilations préalables, on constate que sous .Net l'invocation dynamique et non statique est le mode d'invocation standard. Observons par exemple le code du fichier `TestServiceProxy.cs` généré automatiquement à partir du fichier .asmx (il nous servira par la suite lors de la description des appels asynchrones) :

Fichier TestServiceProxy.cs

```
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;

/// <remarks/>
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(Name="TestServiceSoap", Namespace="http://tempuri.org/")]
public class TestService : System.Web.Services.Protocols.SoapHttpClientProtocol {

    /// <remarks/>
    public TestService() {
        this.Url = "http://localhost/6152/TestService.asmx";
    }

    /// <remarks/>
[System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://tempuri.org/jeTravaillePourLeWeb",
RequestNamespace="http://tempuri.org/", ResponseNamespace="http://tempuri.org/", Use=
System.Web.Services.Description.SoapBindingUse.Literal, ParameterStyle=
System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public string jeTravaillePourLeWeb(string unNom) {
        object[] results = this.Invoke("jeTravaillePourLeWeb", new object[] {
                    unNom});
        return ((string)(results[0]));
    }

    /// <remarks/>
    public System.IAsyncResult BeginjeTravaillePourLeWeb(string unNom, System.AsyncCallback callback,
        object asyncState) {
        return this.BeginInvoke("jeTravaillePourLeWeb", new object[] {
                    unNom}, callback, asyncState);
    }

    /// <remarks/>
    public string EndjeTravaillePourLeWeb(System.IAsyncResult asyncResult) {
        object[] results = this.EndInvoke(asyncResult);
        return ((string)(results[0]));
    }
}
```

On y trouve l'instruction `this.invoke` (« nom de la méthode », « description des paramètres ») caractéristique des invocations dynamiques, puisque l'on passe toute la description du service au moment de l'exécution.

Invocation asynchrone en .Net

En .Net, tout comme en Corba, il est possible, après l'envoi du message, de ne pas bloquer l'expéditeur le temps de l'exécution. L'expéditeur peut alors, s'il le désire, exécuter la suite de son code jusqu'à ce qu'il soit informé du service s'exécutant côté serveur et qu'il en obtienne le résultat. On parle alors d'invocation asynchrone plutôt que synchrone. Nous en donnons un exemple ci-dessous à partir du fichier précédent. Nous l'avons renommé `TestServiceLong.cs`. Son exécution est beaucoup plus longue à cause d'une boucle ridicule qui justifie que le client continue de dérouler son code jusqu'à être informé de la fin du service.

Fichier TestServiceLong.cs

```
<%@ WebService Language="C#" Class="TestService" %>

using System;
using System.Threading;
using System.Web.Services;

public class TestService : WebService {

              [WebMethod]
              public string jeTravaillePourLeWeb (String unNom)
              {
                      return "Salut, " + unNom + jeSuisPriveDansLaClasse();
              }

              private string jeSuisPriveDansLaClasse() {
                      int a = 0;
                      for (int i=0; i<1000000000; i++) { // rallongement idiot de la méthode=boucle
                             ridicule
                                           a++;
                             }
                             return ", je travaille pour le web";
              }
}
```

Fichier TestClientLong.cs

```
using System;
using System.Runtime.Remoting.Messaging;

public class TestClient {
 private static bool bEnd = false;

 public static void CallbackService(IAsyncResult arResult) {

    // afin d'obtenir l'état initial de la proxy
    TestService unTest = (TestService)arResult.AsyncState;
```

```
    // obtenir les résultats du service Web en appelant la méthode End du Proxy
    Console.WriteLine(unTest.EndjeTravaillePourLeWeb(arResult));
    Console.WriteLine("Le service Web vient de se terminer");
    bEnd = true;
 }

 public static void Main() {

    TestService unTest = new TestService();

    // J'appelle le service Web de manière asynchrone - utilisation des délégués
    AsyncCallback acb = new AsyncCallback(CallbackService);
    Console.WriteLine(unTest.BeginjeTravaillePourLeWeb("moi le service",acb,unTest));
    // Je continue comme si de rien n'était
    while (!bEnd) {
      Console.WriteLine("je vaque a mes occupations en attendant");
    }

   }
}
```

Résultats

```
…………………………………………….
…………………………………………….
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
je vaque à mes occupations en attendant
Salut, moi, le service, je travaille pour le web
Le service Web vient de se terminer
```

La réalisation de cet appel asynchrone exige de modifier le code du client en exploitant assez naturellement les deux méthodes `Begin` et `End` générées automatiquement par .Net dans le fichier proxy (voir plus haut).

Mais où sont passés les objets ?

Du côté serveur, l’objet est créé à la volée, le temps de l’exécution du service. Il n’est donc pas nécessaire de désigner un objet précis, enregistré dans un registre comme dans RMI ou la version par défaut de Corba, sur lequel s’exécutera le service. Un nouvel objet est créé pour chaque message qui arrive au serveur, et est détruit à la fin de l’exécution de ce dernier. Cela simplifie grandement les choses et doit pouvoir suffire dans une majorité d’applications. On peut dès lors légitimement se demander l’intérêt qu’il y a à maintenir un objet serveur toute la durée de l’interaction, comme nous l’avons fait en expérimentant RMI et Corba.

S'il est difficile de percevoir ce que cet objet pourrait nous apprendre au début de son activation, il n'en est pas moins vrai que, durant l'interaction, l'objet peut maintenir un ensemble d'informations, du côté serveur, propice à cette interaction. Par exemple, un second envoi de message pourrait ne pas avoir le même effet selon les résultats du premier envoi (résultats enregistrés dans l'état de l'objet serveur). Imaginez un jeu informatique ayant cours sur le réseau ; le comportement de chaque objet, en réponse à un message envoyé par un autre, dépendra de son état. De même, dans une négociation commerciale entre deux objets, les décisions prises par chaque objet lors de cette interaction dépendront de leur connaissance courante de cette négociation.

Une manière de procéder pourrait consister à sauvegarder cet état intermédiaire sur le disque dur (par exemple, dans une base de données). Cependant, vu les temps d'accès disque, cela pourrait considérablement ralentir et alourdir le déroulement de l'application. Il serait plus efficace de retrouver une situation, inhérente à RMI et Corba, de maintien d'information du côté serveur le temps de l'interaction.

Les services Web ne fonctionnent pas directement au-dessus de TCP/IP comme RMI et Corba mais, en raison de leur homogénéisation Web et de leur codage XML, un étage plus haut, au-dessus du protocole HTTP (le protocole de communication Web). Lorsque, au-dessus de ce protocole, une interaction client-serveur doit se dérouler, en maintenant, du côté serveur, des informations sur son état, on invoque souvent la présence de « cookies », comme nous allons le voir. Nous allons reproduire les services Web de l'exemple précédent, en maintenant du côté serveur le nombre de fois que le service est appelé, et en modifiant la réponse du serveur au message en fonction de ce nombre. La nouvelle implémentation asmx du service est la suivante :

Fichier TestServiceAvecMemoire.asmx

```
<%@ WebService Language="C#" Class="TestServiceAvecMemoire" %>

using System;
using System.Web.Services;

[WebService(
    Description = "Un service Web avec mémoire")]
/* il est possible de passer des informations sur la nature du service */

/* il faut maintenant que la classe hérite de WebService pour utiliser l'objet " Session " */
public class TestServiceAvecMemoire : WebService
{
  public TestServiceAvecMemoire() {
    if (nouvelleSession) /* debut de la session */ {
      nouvelleSession = false;
      nbreConnexions = 0;
    }
  }
  /* il est possible également de passer des informations sur la nature de la méthode
     - attention à l'addition de " EnableSession = true ",
     indispensable si cette méthode doit utiliser des informations mémorisées côté serveur */
  [WebMethod(
      Description="Un Service avec memoire",
      EnableSession = true
      )]

  public string jeTravaillePourLeWeb (String unNom) {
    /* le retour sera différent suivant qu'il est invoqué une première fois ou non */
```

```
      if (nbreConnexions == 0) {
        nbreConnexions++;
        return " Salut, " + unNom + jeSuisPriveDansLaClasse();
      }
      else {
        nbreConnexions++;
        return " Encore toi, " + "salut, " + unNom +   jeSuisPriveDansLaClasse();
      }
    }
    private string jeSuisPriveDansLaClasse() {
      return ", je travaille pour le Web";
    }
    private int nbreConnexions { /* méthode d'accès */
      /* utilisation capitale de l'objet Session pour mémoriser l'état du serveur */
      get {
        return (int) Session["nbreConnexions"];
      }
      set  {
        Session["nbreConnexions"] = value;
      }
    }
    private bool nouvelleSession { /* méthode d'accès */
      get {
        if (Session["nouvelleSession"] == null) return true;
        return (bool) Session["nouvelleSession"];
      }
      set {
        Session["nouvelleSession"] = value;
      }
    }
}
```

On relève plusieurs adjonctions par rapport à la version précédente. D’abord, tant dans la définition de la classe service que dans la méthode qui rend le service, des informations supplémentaires peuvent être transmises par l’utilisation d’un attribut `Description`. Ensuite, la classe doit maintenant hériter de `WebService` pour pouvoir utiliser l’objet `Session` qui maintient la mémoire de l’interaction. Cet objet `Session` enregistre un ensemble de variables arbitraires que l’on désigne par `Session["variable"]`.

C’est l’inélégance de cette écriture pour traiter les variables sessions qui nous fait recourir aux méthodes d’accès pour l’attribut entier `nbreConnexions` (qui mémorisera le nombre d’invocations de la méthode) et l’attribut booléen `nouvelleSession` (qui indiquera si oui on non il s’agit d’une nouvelle session). Toute méthode utilisant des informations sur la session doit le signaler dans sa déclaration par : `EnableSession = true`.

Voici maintenant le code du côté client qui doit, lui aussi, par l’addition de l’instruction `unTest.CookieContainer = new CookieContainer()` signaler que cette interaction se fera en maintenant des informations sur l’état du serveur. Les « cookies » apparaissent.

```
using System;
using System.Net;

public class TestClient2 {
```

```
  public static void Main() {
    TestServiceAvecMemoire unTest = new TestServiceAvecMemoire();
    unTest.CookieContainer = new CookieContainer();
    Console.WriteLine(unTest.jeTravaillePourLeWeb(" moi le service "));
    /* le même envoi de message, mais l'effet sera différent */
    Console.WriteLine(unTest.jeTravaillePourLeWeb(" moi le service "));
  }
}
```

Résultat

```
Salut, moi le service, je travaille pour le Web
Encore toi, salut, moi le service, je travaille pour le Web
```

Un annuaire des services XML universel : UDDI

Enfin, existe-t-il dans cette nouvelle infrastructure d'objets distribués, un mode d'organisation et de présentation des services comparable à l'annuaire de Jini ? Oui, car tous les constructeurs se sont mis d'accord sur un mode uniforme de présentation de ces services, dénommé UDDI (Universal Description Discovery and Integration). Tous les services décrits dans le langage WDSL peuvent y être affichés et consultés, ainsi que leur emplacement et la façon de les activer.

Comme dans Jini, dès qu'un de ces services se révèle utile à un client, ce dernier pourra télécharger le proxy du service, de manière, par exemple, à pouvoir communiquer directement avec le serveur responsable du service. La spécification UDDI décrit une série de standards que les fournisseurs de service Web doivent respecter, afin de présenter leur service dans cet annuaire. Dans l'annuaire UDDI, chaque enregistrement contient trois types d'information, décrits sur le modèle des bottins téléphoniques (nom, adresse, contact de l'entreprise), des « Pages jaunes » (catégorie de l'entreprise) et « Pages vertes » (informations plus techniques concernant le service Web). Logiquement centralisé mais physiquement réparti, cet annuaire universel recueille les inscritpions des fournisseurs de services Web et permet à tout un chacun d'effectuer des recherches selon ses besoins.

Services Web vs RMI et Corba

Malgré l'avance prise par Java et le label de standard unique de Corba, il est incontestable que les services web sont en train de damer le pion de leurs concurrents. En premier lieu, du fait de la généralisation d'XML à tout le contenu du Web, que celui-ci soit statique (sites) ou plus dynamique (services). Ensuite, grâce à la facilité et à la rapidité (due à l'automatisation) de mise en œuvre de ces mêmes services – en tout cas en ce qui concerne la plate-forme de développement .Net.

Cependant, les services web ne sont pas à l'abri des critiques.

Le contenu d'un message doit être « parsé » avant de pouvoir s'exécuter. Ce processus est long, et doit en outre faire l'objet d'une standardisation (par exemple : types de données que l'on peut passer en arguments et en retour des méthodes) afin qu'un message XML envoyé par un client soit interprétré correctement par le serveur. C'est loin d'être le cas aujourd'hui avec la multiplication des standards Soap. Les services web sont donc bien plus lents et moins standards que ne l'est Corba aujourd'hui. Comme progrès technologique, on a déjà vu mieux…

Malgré sa dénomination, Soap (Simple Object Access Protocol), le protocole d'envoi et de réception de message, n'est pas du tout orienté objet : rien n'est prévu pour la référence, la sauvegarde ou le maintien de l'état des objets durant une session. Comme nous l'avons vu précédemment, il n'y a pas vraiment d'objets exécutant les

services le temps de l'interaction. Les objets sont les grands absents des services Web. Il semble en fait que l'on soit revenu aux anciens RPC (Remote Procedure Call), par lesquels les applications informatiques, simplement, se sollicitaient mutuellement les exécutions de procédures. Sans doute, le seul véritable avantage des services Web s'avère être un contrôle plus facile de la sécurité, surtout grâce aux pare-feu qui empêchent toute circulation sur des ports autres que http (le port 80 utilisé par les services Web). Corba et RMI se caractérisent par un processus d'allocation de port dynamique, ce qui rend la sécurité des échanges nettement plus délicate à assurer.

Exercices

Exercice 16.1

Expliquez pourquoi la pratique des objets distribués repose dans une large mesure sur la structure syntaxique d'interface.

Exercice 16.2

Comment le compilateur Corba, idl -> C++, traduit-il dans ce langage une interface IDL ?

Exercice 16.3

Expliquez en quoi RMI est plus polymorphique que Corba.

Exercice 16.4

Justifiez l'apport de XML dans le développement des services Web.

Exercice 16.5

Réalisez l'application suivante en Corba : un appareil de retrait d'argent automatique programmé en Java sur un premier ordinateur débite ou crédite des comptes en banque enregistrés sur un second ordinateur. Les dépôts et les retraits d'argent sur ces comptes devront être programmés en C++.

Exercice 16.6

Complétez l'implémentation de l'application RMI suivante, dont l'interface est montrée ci-après : un système automatisé de conversion dans la monnaie locale d'un montant indiqué en euro fonctionne sur un premier ordinateur (car les fluctuations de cours sont connues seulement par ce premier ordinateur). Un second ordinateur permet d'informer des clients sur la valeur dans leur monnaie locale d'une somme en euro.

```
public interface Convertisseur extends java.rmi.Remote {
  public int convertiEnDollar(double enEuro)
    throws java.rmi.RemoteException;
  public int convertiEnFrancais(double enEuro)
    throws java.rmi.RemoteException;
  public int convertiEnLire(double enEuro)
    throws java.rmi.RemoteException;
  public int ...
}
```

Exercice 16.7

Expliquez la démarche suivie tant par Corba que par RMI pour tenter de rendre l’écriture d’applications distribuées très proche de l’écriture d’applications locales.

Exercice 16.8

Expliquez le rôle du stub ou du proxy que l’on retrouve dans toute pratique d’objets distribués.

Exercice 16.9

Expliquez l’apport de Jini par rapport à RMI.

Exercice 16.10

Expliquez la raison du service de nommage, tant en Corba qu’en RMI, et la manière dont les objets sont connus sur le Web à l’aide d’un nom symbolique.

Exercice 16.11

Expliquez pourquoi ce service de nommage a disparu dans le développement des services Web et par quoi il a été remplacé.

Exercice 16.12

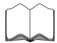
Expliquez la manière dont les services Web compensent la disparition d’un objet maintenu par RMI et Corba le temps de l’interaction client-serveur.

17

Multithreading

Ce chapitre est consacré au multithreading, qui permet à plusieurs objets d'agir de manière simultanée, tout en synchronisant leur accès à des ressources qu'il ne leur est pas possible de partager.

Sommaire : Multithreading — Héritage ou agrégation d'un thread — L'effet du multithreading sur le diagramme de séquence UML — Message synchrone ou asynchrone — La synchronisation des threads



Candidus — Tous ces objets dans un même programme me font penser à une foule d'individus et je me demande comment ils peuvent se satisfaire d'un unique processeur. J'ai l'impression qu'il est devenu impossible de prévoir la suite des instructions qu'il devra exécuter pour s'occuper de tous ces objets.

Doctus — Idéalement, on pourrait associer un processeur à chaque objet et tout deviendrait aussi simple qu'avant. Mais, au risque de te troubler encore un peu plus, ça ne serait pas encore suffisant pour satisfaire pleinement nos objets.

Cand. — Ah non ? Qu'oseraient-ils demander de plus alors ?

Doc. — Un même objet peut être amené à faire plusieurs choses en même temps…

Cand. — Comme toi par exemple lorsque tu parles la bouche pleine ?

Doc. — C'est également le cas lorsque je conduis ma voiture. Mais cela n'a rien d'extraordinaire. Un système d'exploitation se débrouille très bien avec le multitâche pour exécuter plusieurs programmes, en récupérant tes actions sur le clavier et la souris, etc.

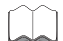
Cand. — Tu veux dire que la programmation objet m'entraînera sur ce terrain glissant des traitements parallèles ?

Doc. — Si tu dois créer des objets serveur, pouvant fournir leur service à plusieurs clients, tu devras te débrouiller pour en satisfaire plusieurs en même temps. Ce sera le cas pour des clients répartis sur un réseau par exemple.

Cand. — On peut toujours les mettre en file d'attente…, mais ce n'est certes pas très efficace.

Doc. — Certains langages nous permettent très facilement de faire fonctionner plusieurs objets en même temps. Il faudra tout de même nous assurer qu'ils ne manipulent pas les mêmes données en même temps. Leur accès devra absolument être synchronisé. On appelle ça le parallélisme, ou multithreading.

Cand. — Je crois bien que je ne vais pas y couper cette fois. Jusque-là, ces traitements parallèles étaient réservés aux programmeurs d'élite. Il ne me reste plus qu'à en devenir un, il me semble…



Replongeons-nous dans notre écosystème Java, et observons la proie et le prédateur se désaltérer ensemble au point d’eau. La classe `Eau` possède une méthode `diminue(int x)`, qui fait diminuer sa quantité de la valeur x jusqu’à l’assécher complètement. La quantité est fixée à une valeur de départ, transmise dans le constructeur, comme indiqué dans le code Java ci-après :

```
public class Eau {
  private int quantite;

  public Eau (int quantite) {
    this.quantite = quantite;
  }
  public void diminue (int decroit) {
    if (quantite > decroit) {
      System.out.println("ok, l'eau diminue");
      quantite -= decroit;
    }
    else {
      quantite = 0;
      System.out.println("zut, il n'y a plus d'eau");
    }
  }
}
```

La méthode `jeBois()` qui permet, tant à la proie qu’au prédateur, de se désaltérer contient une boucle de 100 lapées, le prédateur consommant l’eau deux fois plus vite que la proie. Le code des deux animaux le montre ci-après :

```
public class Proie  {
  private Eau uneEau;

  public Proie(Eau uneEau) {
    this.uneEau = uneEau;
  }
  public void jeBois() {
    for (int i=0; i<100; i++) {
      System.out.println("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
}
public class Predateur {
  private Eau uneEau;

  public Predateur (Eau uneEau) {
    this.uneEau = uneEau;
  }
  public void jeBois(){
    for (int i=0; i<100; i++) {
      System.out.println("le predateur essaie de boire");
      uneEau.diminue(20);
    }
  }
}
```

La classe `Jungle`, quant à elle, crée l'eau, la proie et le prédateur, et ordonne aux deux animaux de se désaltérer.

```
public class Jungle{
  public static void main(String[] args) {
    Eau uneEau = new Eau(1000);
    Proie uneProie = new Proie(uneEau);
    Predateur unPredateur = new Predateur(uneEau);

    uneProie.jeBois();
    unPredateur.jeBois();
  }
}
```

Partie du résultat

```
la proie essaie de boire
OK ! l'eau diminue
la proie essaie de boire
OK ! l'eau diminue
la proie essaie de boire
…
OK ! l'eau diminue
la proie essaie de boire
zut ! il n'y a plus d'eau
le prédateur essaie de boire
zut ! il n'y a plus d'eau
le prédateur essaie de boire
zut ! il n'y a plus d'eau
le prédateur essaie de boire
zut ! il n'y a plus d'eau
le prédateur essaie de boire
…
```

Informatique séquentielle

Hélas, vu la façon dont le code est écrit, et bien que la réalité dépeinte soit celle d'une proie et d'un prédateur se désaltérant de concert, il ne restera plus une goutte d'eau à consommer pour le prédateur, quand la proie en aura terminé avec sa méthode `jeBois()`. Une partie du résultat affiché le montre clairement. En effet, au contraire du monde qui nous entoure, l'informatique fonctionne, dans l'immense majorité des ordinateurs, de manière fondamentalement séquentielle. Le processeur ne peut s'occuper que d'une instruction à la fois. Il doit en avoir terminer avec une, pour attaquer l'autre. On doit cette vision, vieille de 60 ans, et depuis inchangée, à John von Neumann, dont la source d'inspiration première, dans sa conception de l'ordinateur, était tout simplement le cerveau, ordinateur, s'il en est, le plus massivement parallèle qui soit…

Les langages de programmation sont structurés autour de blocs d'instructions, qui s'exécutent, en principe, sans interruption. Ces blocs sont encadrés par des accolades (Python utilise l'indentation) qui délimitent une mini-tâche à accomplir, ainsi que la portée des variables à n'utiliser que durant l'exécution de cette dernière. Si rien n'est fait pour contrer cela, toute mini-tâche s'exécutera d'un bloc, et elle devra obligatoirement se terminer avant qu'une autre ne puisse disposer du processeur. C'est bien pour cela que le prédateur devra attendre les cent lapées de la proie, avant de plonger sa gueule… dans le sable. Si l'OO a pour vocation de mieux coller à la réalité qui nous entoure, il est impératif de lui permettre que des objets, qui exécutent les mini-tâches leur incombant, puissent le faire en parallèle et non plus d'une manière uniquement séquentielle. La proie et le prédateur devraient, comme dans la réalité, pouvoir se partager le point d'eau.

John von Neumann

Nous avons comparé deux modes d'exécution d'un programme, l'un classique, séquentiel, et souvent attaché au nom de von Neumann (en déclarant que le programme tourne sur une machine de type von Neumann), et un autre, parallèle, où plusieurs parties du programme peuvent s'exécuter simultanément sur des processeurs séparés. John von Neumann est considéré, avec Turing, comme le père de l'informatique moderne. Il matérialise l'idée de machine universelle de Turing, par la mise au point de programmes, composés d'une succession d'instructions machine, et installés lors de leur exécution dans la mémoire RAM de l'ordinateur. L'exécution consiste alors en la récupération de ces instructions une à une, en leur installation dans un registre dédié, leur décodage (que font-elles et de quelles opérandes ont-elles besoin) et leur exécution. Une fois cette exécution terminée, on passe à l'instruction suivante. L'universalité et la formidable ubiquité de l'ordinateur sont dues à la variabilité infinie des programmes qu'il est possible d'écrire.

On lit souvent qu'une nouvelle informatique est en train de se défaire de cette mécanique à la von Neumann, victime de cette séquentialité rigide. Cette nouvelle informatique serait davantage inspirée de la biologie (réseau de neurones, informatique génétique…) et mettrait beaucoup plus en relief le parallélisme de processus simples, capables de donner naissance à un deuxième niveau d'observation, à l'émergence de comportements nouveaux, sans équivalent dans le niveau inférieur. Cette informatique serait intrinsèquement plus adaptable, flexible, et capable de surprendre le programmeur par les solutions qu'elle découvrirait, de façon autonome, aux problèmes qui lui seraient posés. La vie artificielle est un de ces nouveaux courants, où l'on utilise massivement l'ordinateur pour reproduire des mécanismes communs aux organismes vivants, et ce dans un substrat autre que biochimique. Dans la vie artificielle, l'ordinateur s'empare du premier rôle, c'est la biologie qui vient à lui plutôt que l'inverse. Il s'agit de faire fonctionner l'ordinateur de manière biologique, d'intégrer algorithmiquement ce qui paraît être les leçons essentielles du vivant, et de les tester par le biais de ce cobaye informatique.

En substance, on oppose donc une informatique à la Von Neumann, séquentielle et trop prédictible, à une informatique plus biologique, parallèle, adaptable et émergente. Or (mais vous nous voyez venir), qui est le père de la vie artificielle ? Von Neumann. Eh oui ! ce même diable d'homme, à la fin de sa vie, s'est attaché à réconcilier biologie et informatique, en inventant les automates cellulaires, les premiers programmes capables d'émergence et d'autoreproduction, et en rédigeant une comparaison détaillée du fonctionnement de l'ordinateur et du cerveau, qui sera publiée à titre posthume (un an après sa mort, en 1958, intitulée *The Computer and the Brain*).

Les grands personnages de l'informatique sont souvent largement ouverts au monde et aux autres sciences (l'informatique n'est somme toute qu'un domaine très récent des sciences et de la technologie). Von Neumann était de ceux-là, un encyclopédiste prodigieux. À 22 ans il est docteur en chimie (tiens ! on retrouve notre chimie) de l'université de Zurich, et à 23 ans docteur en mathématique de l'université de Budapest (dont il est originaire). À 30 ans, il est nommé professeur de mathématiques dans le prestigieux institut de Princeton (aux côtés de Gödel et d'Einstein, qui, lui, avait été refusé à l'entrée de l'université de Zurich). Il y poursuit son étude d'un nouveau formalisme mathématique pour traiter la mécanique quantique (nommé depuis « l'algèbre de von Neumann »), et s'intéresse, dans le même temps, à la formalisation des jeux économiques (dont le fameux dilemme du prisonnier), qu'il aimerait voir se profiler comme les fondements scientifiques de l'économie.

Comme beaucoup de scientifiques de l'époque, la guerre est déterminante dans l'orientation de ses recherches. Il participe activement au projet Manhattan, qui donnera naissance à la bombe atomique, et commence à ressentir la nécessité vitale d'un ordinateur puissant et universel. C'est ainsi qu'à l'époque, il invente la machine de von Neumann, dont un exemplaire est le PC sur lequel vous lisez vos courriers électroniques, éditez vos documents et vous adonnez aux joies de la programmation OO. Dans le milieu des années 1950 , il découvre qu'il est rongé par un cancer et, se sentant sur la fin, interroge un docteur sur la meilleure manière d'occuper ses derniers jours. Celui-ci lui répond banalement de consacrer son temps à ce qu'il y a de plus important pour lui. Il continue de travailler activement sur les armes de mort, mais il se lance dans une autre aventure, plus noble, et diamétralement opposée à la première, élucider la vie. De fait, il consacre ses dernières années au mariage de l'informatique et de la biologie, en bâtissant toute une théorie des automates, automates capables notamment d'un exploit, que l'on pensait être la signature unique des phénomènes vivants, l'autoreproduction. Il ne pourra faire aboutir cette recherche (la même que les chercheurs en vie artificielle, ses héritiers directs, s'échinent à poursuivre), définitivement abattu par l'arme impitoyable – et mortelle à l'époque – d'un automate cellulaire hors contrôle.

Il a révolutionné des domaines scientifiques aussi différents que la logique mathématique, la physique quantique, l'informatique (qu'il invente avec Turing), la cybernétique, la théorie des jeux, l'économie et la biologie évolutionniste, et nous ne vous surprendrons pas en précisant que des membres des services secrets américains ont veillé sur lui jusqu'à ce qu'il rende son dernier souffle afin qu'il ne laisse échapper aucun secret capital sur le fonctionnement du monde et de la technologie. Notre héros fut d'ailleurs, dans les années 1950, un partisan forcené du bombardement nucléaire « préventif » de l'URSS, car il soupçonnait celle-ci de s'être dotée de l'arme nucléaire. Von Neumann, qui aurait eu 100 ans en 2003, fut décidément à l'avant-garde dans tous les champs de la vie et de la pensée humaine.

Multithreading

Le mécanisme dit de « multithreading », et qu'il est possible de directement exploiter dans l'écriture de code Java, C# et Python, rend effectif ce parallélisme des tâches évoqué dans la précédente section. Les programmeurs C++ et PHP 5 doivent, en général, se retourner vers le système d'exploitation ou user de l'un ou l'autre artefact de programmation, pour bénéficier de ce même procédé. Java, C# (ou Visual C++.Net, qui bénéficient des mêmes librairies) et Python en ont fait un utilitaire intégré à même leur syntaxe. C'est la raison pour laquelle, dans les exemples de code à venir, nous nous limiterons à ces trois seuls langages. Le multithreading permet aux blocs d'instruction de s'imbriquer pendant leur exécution.

Le processeur passera la main à un bloc puis à l'autre de manière séquentielle. Les instructions continueront à s'exécuter en séquence (le processeur ne pouvant toujours exécuter qu'une seule instruction à la fois) alors que les blocs, eux, s'exécuteront en parallèle. Chacun de ces blocs, sujet à ce parallélisme d'exécution, sera appelé un « thread ». Comme la figure ci-après le montre, le processeur passera d'un thread à l'autre, sous la responsabilité d'un « gestionnaire de thread », qui saura quand interrompre un thread pour en débuter un autre. Cette gestion du multithreading est généralement laissée au système d'exploitation. Dans la version la plus simple, le processeur se consacre à chacun des threads pendant une même durée. Des stratégies plus fines permettent, par exemple, d'interrompre un thread, quand celui-ci est en attente d'une ressource ou d'un accès périphérique, pour reprendre l'exécution d'un autre.

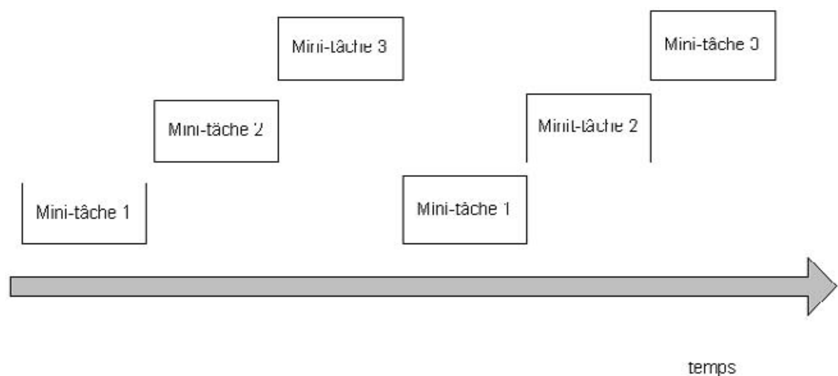


Figure 17-1
Le multithreading répartissant l'occupation du processeur entre trois mini-tâches.

Malgré la volonté de Java, de Python, et, dans une moindre mesure, de C# (très dépendant du ou des Windows) de s'émanciper autant que faire se peut des plates-formes sur lesquelles ils s'exécutent (pour que les applications tournent de la même manière, quelle que soit cette plate-forme), le résultat d'un programme intégrant un mécanisme de multithreading pourra largement varier d'une plate-forme à l'autre. Ce mécanisme est,

à une moindre échelle (car à l’intérieur d’un seul et même programme), la réplique exacte du mécanisme de multitâche, présent dans tous les systèmes d’exploitations actuels.

Le multithreading, une réplique à moindre échelle du multitâche

Le multitâche vous permet d’utiliser différentes applications, Word, Firefox, votre jeu favori, votre système de courrier électronique… avec la même apparence de simultanéité que la proie et le prédateur s’abreuvant de concert. Le multitâche et sa version réduite, le multithreading, s’exécutent de la même façon. Cela se déroule de la manière suivante : le gestionnaire entame une première mini-tâche (thread), en exécute quelques instructions, puis est interrompu pour donner la main à une deuxième mini-tâche. Avant de passer la main, il mémorise tout ce qui lui est nécessaire pour pouvoir reprendre l’exécution de la première mini-tâche, à peine délaissée, par exemple, l’adresse de l’instruction suivante à exécuter ou les adresses de fichiers et autres périphériques avec lesquels cette mini-tâche interagissait. En substance, il mémorise tout le contexte d’exécution de cette mini-tâche. Après avoir donné un peu de son temps à toutes les mini-tâches qui s’exécutent à la queue leu leu et pourtant parallèlement, il revient à la première, en commençant, avant toute chose, par restituer son contexte d’exécution.

Implémentation en Java

Nous allons donc permettre à la proie et au prédateur de boire ensemble, pacifiquement, la soif l’emportant sur la faim. Pour ce faire, nous allons, tant en Java qu’en C# et Python, implémenter ce mécanisme de multithreading, pour que la méthode `jeBois()` de la proie et celle du prédateur puissent s’exécuter simultanément. Commençons par Java :

```
public class Proie extends Thread { /* La proie devient un Thread */
  private Eau uneEau;

  public Proie(Eau uneEau) {
    this.uneEau = uneEau;
  }
  public void jeBois(){
    for (int i=0; i<100; i++){
      System.out.println("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
  public void run() { /* méthode héritée de la classe Thread et à redéfinir */
    jeBois();
  }
}
public class Predateur extends Thread {
  private Eau uneEau;

  public Predateur (Eau uneEau) {
    this.uneEau = uneEau;
  }
  public void jeBois(){
    for (int i=0; i<100; i++) {
      System.out.println("le predateur essaie de boire");
      uneEau.diminue(20);
    }
  }
  public void run() {
    jeBois();
```

```
    }
  }
  public class Jungle {
    public static void main(String[] args){
      Eau uneEau = new Eau(1000);
      Proie uneProie = new Proie(uneEau);
      Predateur unPredateur = new Predateur(uneEau);
      uneProie.start();       /* démarrage du premier thread */
      unPredateur.start();    /* démarrage du deuxième thread */
    }
  }
```

Voici une partie du résultat

```
le prédateur essaie de boire
la proie essaie de boire
OK ! l'eau diminue
la proie essaie de boire
OK ! l'eau diminue
la proie essaie de boire
OK ! l'eau diminue
OK ! l'eau diminue
la proie essaie de boire
le prédateur essaie de boire
OK ! l'eau diminue
OK ! l'eau diminue
le prédateur essaie de boire
OK ! l'eau diminue
la proie essaie de boire
le prédateur essaie de boire
…
le prédateur essaie de boire
zut ! il n'y a plus d'eau
zut ! il n'y a plus d'eau
la proie essaie de boire
le prédateur essaie de boire
zut ! il n'y a plus d'eau
zut ! il n'y a plus d'eau
la proie essaie de boire
```

Nous constatons que la proie et le prédateur exécutent leur méthode `jeBois()` simultanément, sans que nous ayons un quelconque contrôle sur la durée que chacun peut consacrer à cela. Cette répartition du processeur entre les deux méthodes qui doivent s'exécuter ensemble est la seule responsabilité du système d'exploitation. Les deux animaux boivent ensemble, chacun pendant une durée, apparemment aléatoire (nous verrons par la suite comment répartir cette durée de façon plus uniforme), et se retrouvent, en tous les cas, de la même manière, quand l'eau est épuisée, le bec dans l'eau… euh ! non, dans le sable.

Il y a plusieurs manières d'implémenter le multithreading en Java. Ici, nous avons fait, tant de la proie que du prédateur, un « thread », en faisant hériter les deux classes de la classe `Thread`. La seule méthode qu'il nous importe de récupérer de la classe `Thread`, et de redéfinir, est la méthode `run()`, dont le corps d'instruction se compose de celles que nous désirons exécuter en parallèle avec d'autres. Les deux blocs d'instructions sont, en fait, les deux méthodes `jeBois()`, car la redéfinition des deux `run()` se limite à appeler ces méthodes.

Une autre manière de faire, plus fréquente, est l'utilisation de l'interface `Runnable` car souvent le seul héritage possible aura été déjà « épuisé » par une autre super-classe. Nous retrouvons ci-dessous le code de la classe `Proie` implémentant l'interface `Runnable` qui oblige, cette fois, à la redéfinition de la méthode `run`. Aussi, le thread associé à la `Proie` doit maintenant se trouver explicitement crée et agrégé dans la classe en question :

```
class Proie implements Runnable {
  private Eau uneEau;
  Thread tProie; // il faut maintenant agréger le thread dans la classe

  public Proie(Eau uneEau) {
    this.uneEau = uneEau;
    tProie = new Thread(this); // this réfère à Runnable
    tProie.start(); // Il faut maintenant le démarrer ici
  }
  public void jeBois(){
    for (int i=0; i<100; i++){
      System.out.println("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
  public void run() { /* méthode héritée de l'interface et à obligatoirement redéfinir */
    jeBois();
  }
}
```

Lorsqu'un thread est créé (ici la proie et le prédateur héritant de la classe `Thread`, les deux threads seront automatiquement créés quand les deux animaux le seront), il est d'abord dans un état dormant. Il faut le lancer par l'entremise de la méthode `start()`, afin de le rendre disponible pour le gestionnaire du « multithreading », c'est-à-dire sélectionnable pour occuper le processeur. Un thread pourra, par la suite, être supprimé ou suspendu, repris ou détruit, par des méthodes et une gestion appropriées (Attention ! Les dernières versions de Java se sont beaucoup modifiées en ce qui concerne les mécanismes de suspension et de redémarrage des threads).

Implémentation en C#

En C#

```
using System.Threading;
class Eau{
  private int quantite;

  public Eau (int quantite){
    this.quantite = quantite;
  }
  public void diminue (int decroit){
    if (quantite > decroit){
```

```
      Console.WriteLine("ok, l'eau diminue");
      quantite -= decroit;
    }
    else{
      quantite = 0;
      Console.WriteLine("zut, il n'y a plus d'eau");
    }
  }
}
class Predateur {
  private Eau uneEau;
  private Thread unThread;  /* le thread est agrégé */

  public Predateur(Eau uneEau){
    this.uneEau = uneEau;
    ThreadStart unTs = new ThreadStart(jeBois); /* un délégué est créé sur la méthode à associer
   ➥au thread */
    unThread = new Thread(unTs);  /* le thread est créé en lui passant le délégué */
  }
  public void lanceThread(){
    unThread.Start(); /* démarrage du thread */
  }
  public void jeBois(){
    for (int i=0; i<100; i++){
      Console.WriteLine("le predateur essaie de boire");
      uneEau.diminue(20);
    }
  }
}
class Proie {
  private Eau uneEau;
  private Thread unThread;

  public Proie(Eau uneEau){
    this.uneEau = uneEau;
    unThread = new Thread(new ThreadStart(jeBois));
  }
  public void jeBois(){
    for (int i=0; i<100; i++){
      Console.WriteLine("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
  public void lanceThread(){
    unThread.Start();
  }
}
```

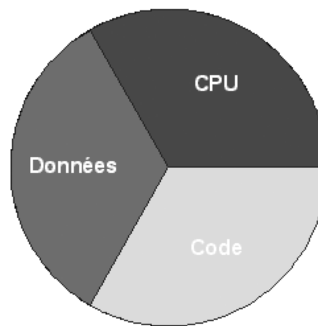
```
public class Jungle{
  public static void Main(){
    Eau uneEau = new Eau(1000);
    Proie uneProie = new Proie(uneEau);
    Predateur unPredateur = new Predateur(uneEau) ;
    uneProie.lanceThread();
    unPredateur.lanceThread();
  }
}
```

Le résultat sera évidemment le même que celui du code Java, à l'implémentation près du multithreading, et de son interaction avec l'exécutable C# sur le système d'exploitation que vous utilisez. Comme première différence importante, nous voyons, alors que Java force plutôt la pratique d'héritage pour récupérer les utilitaires du multithreading, que C# , quant à lui, adopte la pratique de composition ou d'agrégation forte. Nous avons vu dans le chapitre 11 consacré à l'héritage que, les effets étant à ce point semblables, le choix de l'une ou de l'autre dépendait en général de la sémantique du problème.

Mais, ici, hérite-t-on d'un thread ou contient-on un thread ? Impossible de se prononcer ? Les deux visions se valent. Une autre différence importante est la manière dont le bloc d'instructions est associé au thread. On dit des threads qu'ils sont composés de trois compartiments, comme le montre la figure ci-après. Les trois compartiments sont : le processeur sur lequel le thread s'exécute, les données qu'il manipule (par exemple, les attributs de l'objet, instance de la classe dans laquelle le thread est défini), et le corps d'instructions, la mini-tâche, associée au thread. En Java, cette mini-tâche est transmise au thread par la redéfinition de la méthode `run()`. Cette méthode n'a donc d'autre raison d'être que de déclarer, pour un thread, le corps d'instructions qui le constitue.

Figure 17-2

Les trois compartiments d'un thread.



Cette même transmission se fait en C#, en recourant à une nouvelle structure de données, propre au C# (et .Net en général), et que l'on appelle les « délégués ». Un délégué – dont la création et le mode d'utilisation ressemblent à s'y méprendre aux interfaces (mais qui ne contiendrait qu'une et une seule méthode à concrétiser) – produit des instances associées, qui se limitent à n'être que des référents de fonction. Un délégué pointe sur une méthode, au même titre qu'un référent pointe sur un objet. Nous préciserons et reviendrons sur ce mécanisme de délégué inhérent à la plate-forme .Net pas plus tard que dans le prochain chapitre. Ici, le délégué utilisé est de type `ThreadStart`, et sa seule raison d'être est de pointer sur le bloc d'instructions à associer, en C#, à un thread. Comme en Java, les deux threads sont démarrés par la méthode `Start()` (avec une majuscule initiale… juste pour embêter son monde...).

Implémentation en Python

```
from threading import Thread #un import indispensable

class Eau:
    __quantite=0
    def __init__(self,quantite):
        self.__quantite=quantite
    def diminue(self,decroit):
        if self.__quantite>decroit:
            print "ok, l'eau diminue"
            self.__quantite=decroit
        else:
            self.__quantite=0
            print "zut, il n'y a plus d'eau"

class Proie(Thread):
    __uneEau=None
    def __init__(self,uneEau):
        self.__uneEau=uneEau
        Thread.__init__(self)
    def jeBois(self):
        i=0
        while i<100:
            print "la proie essaie de boire"
            self.__uneEau.diminue(10)
            i+=1
    def run(self):
        self.jeBois()

class Predateur(Thread):
    __uneEau=None
    def __init__(self,uneEau):
        self.__uneEau=uneEau
        Thread.__init__(self)
    def jeBois(self):
        i=0
        while i<100:
            print "le predateur essaie de boire"
            uneEau.diminue(20)
            i+=1
    def run(self):
        self.jeBois()

uneEau=Eau(100)
uneProie=Proie(uneEau)
unPredateur=Predateur(uneEau)
uneProie.start()
unPredateur.start()
```

Dès l'import du module `threading`, l'implémentation Python est très proche de l'implémentation Java avec héritage de la classe `Thread`, définition de la méthode `run` et envoi du message `start` pour débuter les deux threads.

L’impact du multithreading sur les diagrammes de séquence UML

Il est utile de refaire un petit détour par UML et ses diagrammes de séquence, de manière à différencier les diagrammes correspondant aux deux pratiques, sans et avec multithreading.

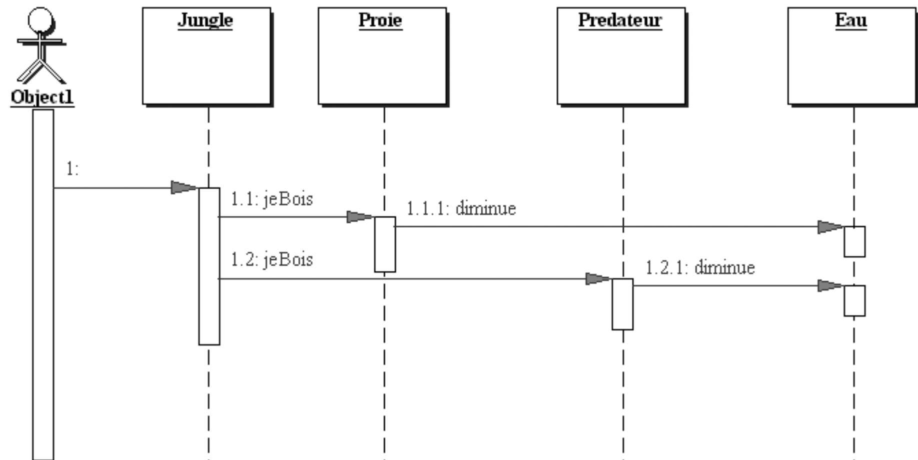


Figure 17-3
Diagramme de séquence de la proie et du prédateur se désaltérant sans *multithreading.*

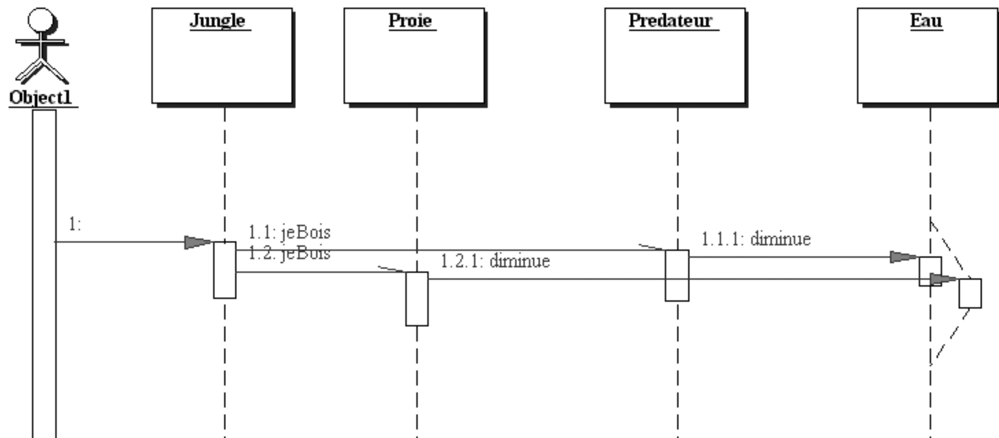


Figure 17-4
Diagramme de séquence de la proie et du prédateur se désaltérant avec *multithreading.*

Lorsqu’un message est déclenché de manière synchrone, la flèche qui le représente dans le diagramme de séquence est complète, et les « rectangles de temporalité » s’ajustent en fonction. Cela veut dire qu’en l’absence de multithreading, le premier message `jeBois`, envoyé à la proie, doit se terminer avant que le second message `jeBois`, envoyé cette fois au prédateur, ne puisse débuter. En revanche, lorsqu’un message est déclenché de manière asynchrone, la flèche qui le représente est différemment dessinée, et les « rectangles de temporalité » n’ont plus à s’ajuster en fonction.

En fait, dans ce cas, le déroulement de l'expéditeur du message n'est plus conditionné par le déroulement du destinataire. Les deux objets vivent leur propre vie, indépendamment l'un de l'autre. L'eau exécute deux fois le même message, de manière parallèle, comme le diagramme de séquence l'indique également. On conçoit que ce même type de diagramme de séquence puisse se retrouver lors de la réalisation d'applications distribuées, dans la mesure où l'objet expéditeur et l'objet destinataire sont actifs sur des processeurs différents.

Du multithreading aux applications distribuées

Du multithreading aux applications distribuées, il n'y a qu'un pas, car deux objets, exécutant parallèlement deux blocs d'instructions séparés, pourraient idéalement se trouver sur deux processeurs différents. Lorsqu'un message est envoyé à travers Internet, il n'y a *a priori* aucune raison pour que son expéditeur se tourne les pouces, en attendant qu'il finisse de s'exécuter. De fait, tous les protocoles d'objets distribués autorisent, d'une manière ou d'une autre, des envois de messages asynchrones, qui n'interrompent pas le déroulement de l'exécution de l'expéditeur. Quant à l'illustration du plus récent de ces protocoles, nous avons vu dans le chapitre précédent comme il est simple de réaliser, par exemple, des services Web asynchrones.

Choisir entre un envoi synchrone ou asynchrone ne dépend *in fine* en rien de l'architecture informatique qui supporte cet envoi de message, mais bien de la seule logique de l'application à exécuter. L'objet expéditeur a- t-il, oui ou non, besoin d'une réponse du destinataire pour aller de l'avant avec son flot d'instructions ? Si la réponse est non, il pourra continuer à s'exécuter, soit sur un thread qui lui est propre, dans le cas d'une informatique purement séquentielle, soit sur un processeur qui lui est propre, dans le cas d'une informatique qui peut être parallélisée au moins sur deux processeurs comme nous l'avons vu dans le chapitre précédent.

Dans la pratique, les raisons d'exploiter le multithreading sont très nombreuses. Les applications de type e-commerce, où plusieurs clients veulent bénéficier simultanément des services d'un serveur, ne peuvent se réaliser autrement qu'en multithreading : un client, un thread. Il sera important, dans ce cas, de synchroniser les multiples interactions client-serveur afin d'établir, avec prudence, à quel moment de l'interaction on peut passer d'un client à l'autre. Nous verrons une manière d'y parvenir par la suite.

Les animations graphiques, si fréquentes dans les pages web, exigent généralement un thread qui leur est propre, afin que leur déroulement continu (souvent, une boucle infinie opérant sur un vecteur d'images) n'empêche pas de regagner le contrôle du processeur pour d'autres interactions. Autrement dit, il ne faudrait pas que ce livre défilant sous toutes ses coutures, sur l'écran du site web d'Eyrolles, vous empêche d'entrer dans le formulaire votre numéro de carte de crédit (message subliminal…).

Enfin, les architectures de vos ordinateurs étant de plus en plus souvent multiprocesseur, il est clair que la meilleure manière de garantir l'utilisation en parallèle de leurs processeurs à l'exécution de vos programmes est d'exploiter la possibilité du multithreading.

Des threads équirépartis

Nous aimerions reprendre un certain contrôle sur la façon dont la proie et le prédateur consomment l'eau. Idéalement, la proie puis le prédateur, et ce de façon alternée, devraient pouvoir se désaltérer. Il y a plusieurs façons d'y arriver. Une première, très simple, est d'utiliser la méthode `yield()` dans la boucle de la méthode `jeBois()`. Cette méthode interrompt le thread, juste le temps pour le gestionnaire de pouvoir donner la main

à un autre thread. Comme il n’y en a que deux ici, c’est automatiquement le second qui prendra le contrôle du processeur et, ainsi de suite, de façon parfaitement équitable.

Une autre manière est de jouer sur la priorité des threads. On peut attribuer aux threads différents niveaux de priorité, qui permettront à certains de mobiliser plus souvent le processeur que d’autres. Une dernière manière est de recourir à la méthode `sleep()`, comme le font les trois codes Java, C# et Python qui suivent :

En Java

```
public void jeBois() {
  for (int i=0; i<100; i++) {
    System.out.println("la proie essaie de boire");
    uneEau.diminue(10);
    try {
      sleep(100); /* le sleep est dans un try-catch, en argument : le nombre de millisecondes */
    } catch (Exception e) {}
  }
}
```

En C#

```
public void jeBois() {
  for (int i=0; i<100; i++) {
    Console.WriteLine("la proie essaie de boire");
    uneEau.diminue(10);
    Thread.Sleep(100); /* le sleep est statique, et ne peut être appelé qu'à partir de sa classe */
  }
}
```

En Python

```
import time # un import additionnel est nécessaire

  def jeBois(self):
            i=0
            while i<100:
                    time.sleep(0.01); #à indiquer en secondes et non pas en millisecondes
                    print "la proie essaie de boire"
                    self.__uneEau.diminue(10)
                    i+=1
    }
  }
```

Python se particularise en ceci que les contrôles des threads ne se fait pas directement à partir de ces threads eux-mêmes, mais à partir d’objets extérieurs, ici l’objet `time`, par exemple.

Résultats

```
le prédateur essaie de boire
OK ! ça marche
la proie essaie de boire
OK ! ça marche
```

```
le prédateur essaie de boire
OK ! ça marche
la proie essaie de boire
OK ! ça marche
le prédateur essaie de boire
OK ! ça marche
la proie essaie de boire
OK ! ça marche
le prédateur essaie de boire
OK ! ça marche
la proie essaie de boire
OK ! ça marche
le prédateur essaie de boire
OK  ça marche
la proie essaie de boire
```

Il y a bien une alternance parfaite entre la proie et le prédateur.

La méthode `sleep()` force le thread à se mettre en repos pendant la durée, exprimée en milliseconde, transmise comme argument de la méthode. Automatiquement, elle incite le processeur à ré-activer un autre thread, puisque le précédent est mis en veille. Cette méthode permet également d'ajuster la vitesse d'exécution du thread, car plus la durée de mise en veille est importante, plus le temps mis par l'exécution du thread sera rallongé. Dans une application graphique animée, cela permet d'ajuster la vitesse de l'animation. La méthode `Sleep()` est déclarée statique dans la classe `Thread` (en C#, elle ne peut s'appeler qu'à partir de sa classe, car C#, contrairement à Java, ne permet pas à une méthode statique d'être appelée à partir d'un objet). En Python, le `sleep` est appelé de façon extérieure au thread, directement sur un objet `time`.

On conçoit aisément que la méthode ne s'applique systématiquement qu'au seul objet thread en activité et que, dès lors qu'il n'est pas nécessaire de préciser l'objet concerné, il est logique que cette méthode soit définie comme statique ou appelée de l'extérieur. Une autre différence entre les codes C#, Python et Java tient à ce que ce dernier force beaucoup plus l'utilisation de la gestion d'exception. Dans le cas du `sleep()`, qui peut en effet mal se passer, vu l'intimité de cette méthode avec le fonctionnement du processeur, cette gestion est obligatoire en Java et facultative en C# ou Python. Comme c'est le cas pour ce `sleep()`, C# et Python se comportent souvent comme C++, accordant leur confiance au programmeur pour qu'en son âme et conscience, celui-ci décide s'il y a lieu ou non de recourir à la gestion d'exception.

Synchroniser les threads

Alors que l'eau peut être consommée simultanément par la proie et le prédateur (cette ressource est suffisamment étendue pour qu'ils restent à distance l'un de l'autre), nous considérerons qu'il n'en va plus de même pour la plante. Si la plante est occupée par la proie ou par le prédateur, un des deux devra attendre son tour pour pouvoir s'y ressourcer. Ce que vit la plante ici est le quotidien, dans les applications d'entreprises, des bases de données, quand elles sont accédées par plusieurs utilisateurs à la fois. Si les modifications apportées par le premier utilisateur peuvent avoir un impact sur celles que cherche à réaliser un second, il est primordial de laisser le premier aller jusqu'au bout de sa manœuvre.

En l'absence d'un tel souci de synchronisation, les résultats deviendront parfaitement incohérents. Supposez par exemple des réservations de billet d'avion dans une même base de données. Il est indispensable qu'une première réservation se termine entièrement avant d'en lancer une deuxième (à partir d'un lieu physique différent

et indépendant du premier). Si ce n'était pas le cas et si le guichetier confirme sa réservation au passager sans avoir mis à jour la base de données, cette même réservation pourrait être effectuée ailleurs par un autre guichetier. Dans un cas semblable, c'est en effet la base de données qui doit s'occuper de la synchronisation de son accès par les différents programmes qui cherchent à la modifier. Les longues attentes que vous devez parfois subir dans les aéroports pour la réservation informatique d'une place d'avion sont la conséquence de ce souci de synchronisation. Attentes qui valent toujours mieux que de retrouver sur vos genoux dans l'avion le passager qui vous suivait dans la file.

Voyons comment, en Java puis en C# et finalement en Python, la plante bloque son accès à la proie ou au prédateur, dès que l'un des deux s'y ressource.

En Java

```
public class Plante {
  private int quantite;

  public Plante (int quantite){
    this.quantite = quantite;
  }
  public void diminue (int decroit){
    if (quantite > decroit) {
      System.out.println("ok, la plante diminue");
      quantite -= decroit;
    }
    else {
      quantite = 0;
      System.out.println("zut, il n'y a plus de plante");
    }
  }
}
public class Proie extends Thread {
  private Eau uneEau;
  private Plante unePlante;

  public Proie(Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
  }
  public void jeBois(){
    for (int i=0; i<100; i++){
      System.out.println("la proie essaie de boire");
      uneEau.diminue(10);
      try{
        sleep(10);
      } catch (Exception e) {}
    }
  }
  public void jeMange(){
    synchronized (unePlante) { /* la plante devient inaccessible par qui que ce soit */
      for (int i=0; i<25; i++) {
        System.out.println("la proie essaie de manger");
        unePlante.diminue(10);
      }
    }
  }
```

```
  public void run() {
    jeBois();
    jeMange();
  }
}
public class Predateur extends Thread {
  private Eau uneEau;
  private Plante unePlante;

  public Predateur (Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
  }
  public void jeBois() {
    for (int i=0; i<100; i++) {
      System.out.println("le predateur essaie de boire");
      uneEau.diminue(20);
      try {
        sleep(100);
      } catch(Exception e) {}
    }
  }
  public void jeMange() {
    synchronized (unePlante) { /*la plante devient inaccessible par qui que ce soit */
      for (int i=0; i<25; i++) {
        System.out.println("le predateur essaie de manger");
        unePlante.diminue(20);
      }
    }
  }
  public void run() {
    jeBois();
    jeMange();
  }
}
```

En C#

```
class Plante {
  private int quantite;

  public Plante (int quantite) {
    this.quantite = quantite;
  }
  public void diminue (int decroit) {
    if (quantite > decroit) {
      Console.WriteLine("ok, la plante diminue");
      quantite -= decroit;
    }
    else {
      quantite = 0;
      Console.WriteLine("zut, il n'y a plus de plante");
    }
  }
```

```
}
class Predateur {
  private Eau uneEau;
  private Thread unThread;
  private Plante unePlante;

  public Predateur(Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
    ThreadStart unTs = new ThreadStart(jeConsomme);
    unThread = new Thread(unTs);
  }
  public void lanceThread() {
    unThread.Start();
  }
  public void jeConsomme() {
    jeBois();
    jeMange();
  }
  public void jeBois(){
    for (int i=0; i<100; i++)
    {
      Console.WriteLine("le predateur essaie de boire");
      uneEau.diminue(20);
      Thread.Sleep(100);
    }
  }
  public void jeMange() {
    Monitor.Enter (unePlante); /* la plante devient inaccessible */
    for (int i=0; i<25; i++) {
      Console.WriteLine("le predateur essaie de manger");
      unePlante.diminue(10);
    }
    Monitor.Exit (unePlante);
    /* une manière alternative est d'utiliser "lock(unePlante)"
     * et de mettre entre accolades le bloc d'instructions qui suit,
     * le "lock" en C# devient l'équivalent du "synchronized" en Java */
  }
}
class Proie  {
  private Eau uneEau;
  private Thread unThread;
  private Plante unePlante;

  public Proie(Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
    unThread = new Thread(new ThreadStart(jeConsomme));
  }
  public void jeConsomme() {
    jeBois();
    jeMange();
  }
```

```
  public void jeBois() {
    for (int i=0; i<100; i++) {
      Console.WriteLine("la proie essaie de boire");
      uneEau.diminue(10);
      Thread.Sleep(100);
    }
  }
  public void jeMange() {
    Monitor.Enter (unePlante); /* la plante devient inaccessible */
    for (int i=0; i<25; i++) {
      Console.WriteLine("la proie essaie de manger");
      unePlante.diminue(10);
    }
    Monitor.Exit (unePlante);
  }
  public void lanceThread() {
    unThread.Start();
  }
}
```

Résultats (en partie) : la proie essaie de boire

```
zut ! il n'y a plus d'eau
la proie essaie de boire
zut ! il n'y a plus d'eau
la proie essaie de manger
le prédateur essaie de boire
OK ! la plante diminue
zut ! il n'y a plus d'eau
la proie essaie de manger
OK ! la plante diminue
la proie essaie de manger
OK ! la plante diminue
…
zut ! il n'y a plus d'eau
le prédateur essaie de boire
zut ! il n'y a plus d'eau
le prédateur essaie de manger
OK ! la plante diminue
le prédateur essaie de manger
OK ! la plante diminue
…
```

D'abord, nous avons rajouté dans les codes la classe `Plante`, qui ressemble, comme deux gouttes d'eau, à la classe du même nom. Dans le code C#, nous avons dû créer une méthode `jeConsomme()`, qui joint les deux méthodes `jeBois()` et `jeMange()` dans le corps d'instructions à associer au thread. La synchronisation se fait en Java par l'addition dans la méthode `jeMange()` d'un bloc `synchronized(laPlante){}`, qui entoure la partie de code qui ne peut occuper la plante que de manière exclusive. Il est important de comprendre que l'arbitre du mécanisme de synchronisation entre threads est la ressource elle-même, cette même ressource qu'il est impossible de partager. Ici, c'est de la plante qu'il s'agit.

Cette synchronisation s'opère car le thread confie à la ressource un « sémaphore » qui, le temps d'une partie de son exécution, n'accepte pas que soit partagée cette ressource. Si un autre thread veut utiliser cette même ressource (ici, toujours la plante), il devra attendre que celle-ci ait libéré le sémaphore qu'elle a en sa possession. Cette libération effectuée, un autre thread pourra déclencher la méthode utilisant cette ressource, quitte à ce qu'il redonne, si lui aussi cherche à synchroniser cet accès, le sémaphore à peine récupéré à la ressource. Un rôle analogue à ce sémaphore serait celui de la lampe rose ou rouge décorant l'entrée des maisons closes d'antan et signalant au client à peine arrivé qu'il se doit d'attendre son tour. Moins grivois est la clé des toilettes que certains barmans parisiens (réputés dans le monde entier pour leur rare amabilité) gardent au comptoir et confient aux clients qui consomment uniquement. En l'absence de cette clé au comptoir, la ressource est bien occupée et exige d'être libérée avant de pouvoir être réoccuppée par un prochain client.

La synchronisation peut se faire sur une méthode tout entière, par exemple, en Java, en ajoutant `synchronized` dans la signature de la méthode si c'est simplement la méthode dans son entièreté qui bloque l'accès à l'objet sur lequel elle s'exécute. Ici, la méthode étant déclarée dans la proie et le prédateur, nous n'avons pas synchronisé la méthode entière, car ce sont alors à la proie et au prédateur auxquels nous aurions bloqué l'accès. L'équivalent en C# est obtenu, soit en remplaçant `synchronized` par `lock`, soit en encadrant la partie de code à synchroniser par `Monitor.Enter()` et `Monitor.Exit()`.

Sémaphore

La détention par la ressource à synchroniser d'un « sémaphore », le temps de son utilisation, de telle manière que d'autres threads n'y aient plus accès, est la base de cette pratique de synchronisation. Cela permet à la ressource, elle-même, d'arbitrer son utilisation.

On voit qu'à la différence de ce qui se passe pendant que la proie boit, pendant qu'elle mange, la plante lui est réservée exclusivement. Le prédateur ne pourra s'attaquer à la plante que lorsque la proie en aura terminé. Remarquez qu'il persiste une incertitude sur qui, de la proie ou du prédateur, attaquera la plante le premier, car cela dépend duquel des deux threads en aura fini le plus vite avec l'eau.

En Python

```
 import time
import threading
from threading import Thread

class Plante(Thread):
      __quantite=0
      lock=None
      def __init__(self,quantite):
               self.__quantite=quantite
               self.lock = threading.Lock() #obtention d'un objet Lock fournit par le module threading
      def diminue(self,decroit):
               if self.__quantite>decroit:
                         print "ok, la plante diminue"
                         self.__quantite-=decroit
               else:
                         self.__quantite=0
                         print "zut, il n'y a plus de plante"
```

```
class Eau:
      __quantite=0
      def __init__(self,quantite):
               self.__quantite=quantite
      def diminue(self,decroit):
               if self.__quantite>decroit:
                         print "ok, l'eau diminue"
                         self.__quantite-=decroit
               else:
                         self.__quantite=0
                         print "zut, il n'y a plus d'eau"

class Proie(Thread):
      __uneEau=None
      __unePlante=None
      def __init__(self,uneEau,unePlante):
               self.__uneEau=uneEau
               self.__unePlante=unePlante
               Thread.__init__(self)
      def jeBois(self):
               i=0
               while i<100:
                         time.sleep(0.01)
                         print "la proie essaie de boire"
                         self.__uneEau.diminue(10)
                         i+=1
      def jeMange(self):
           self.__unePlante.lock.acquire() #bloque l’accès à la plante
               i=0
               while i<25:
                         print "la proie essaie de manger"
                         self.__unePlante.diminue(10)
                         i+=1
               self.__unePlante.lock.release() #libère l’accès à la plante
      def run(self):
               self.jeBois()
               self.jeMange()

class Predateur(Thread):
      __uneEau=None
      __unePlante=None
      def __init__(self,uneEau,unePlante):
               self.__uneEau=uneEau
               self.__unePlante=unePlante
               Thread.__init__(self)
      def jeBois(self):
               i=0
               while i<100:
                         time.sleep(0.001)
                         print "le predateur essaie de boire"
                         uneEau.diminue(20)
                         i+=1
```

```
      def jeMange(self):
           self.__unePlante.lock.acquire()
                i=0
                while i<25:
                           print "le predateur essaie de manger"
                           self.__unePlante.diminue(20)
                           i+=1
                self.__unePlante.lock.release()
      def run(self):
                self.jeBois()
                self.jeMange()

uneEau=Eau(100)
unePlante=Plante(50)
uneProie=Proie(uneEau,unePlante)
unPredateur=Predateur(uneEau,unePlante)
uneProie.start()
unPredateur.start()
```

En Python, l'équivalent des codes précédents se fait par l'obtention d'un objet de type `Lock` sur la plante dont l'état peut être soit « verrouillé », soit « déverrouillé ». La méthode `acquire()`, appelée sur l'objet `Lock`, fait que le thread appelant verrouille la plante pour tout autre thread, alors que la méthode `release` déverrouille la plante pour les autres threads.

D'autres procédés, plus fins encore, de synchronisation entre plusieurs threads existent. Ainsi, un thread peut décider de se suspendre et de reprendre son cours après qu'un autre en a terminé (en Java, C # et Python, la méthode `join()` s'en occupe et réalise une espèce de « jonction » entre des threads). Le code C# qui suit permet de comprendre l'utilité de la méthode `join`, en différenciant le résultat obtenu en sa présence et en son absence.

```
using System;
using System.Threading;

public class Printable {
   private String s;

   public Printable(String s) {
     this.s = s;
   }

   public void Print() {
       Thread.Sleep(10000);
       Console.WriteLine(s);
   }
}

public class Testeur {
   public static void Main() {
      Printable p = new Printable("Hello Hugues");
      Thread t1 = new Thread(new ThreadStart(p.Print));
      Console.WriteLine("Debut du thread");
```

```
        t1.Start();
        Console.WriteLine("Joindre les threads");
        t1.Join(); // différencier le résultat avec et sans la jointure
        Console.WriteLine("Les threads sont joints");
    }
}



Résultat :
Debut du thread
Joindre les threads
Hello Hugues
Les threads sont joints
```

Les deux dernières lignes seront inversées en l'absence de l'instruction `t1.Join()`. La raison en est que la dernière instruction du code, écrivant à l'écran « les threads sont joints » sera, en présence de la jointure, exécutée après la fin du thread `t1`. En fait, nous sommes bien ici en présence de deux threads, celui lié au code principal et `t1`. La présence du `join`, agissant sur le thread `t1`, force le code principal à s'interrompre le temps de l'exécution de `t1`, avant de pouvoir reprendre son cours.

De même, un thread peut décider de surseoir à son exécution en attendant qu'un autre lui permette de reprendre son cours (le couplage `wait-notify` en Java et Python et `wait-pulse` en C#). Mais on aura compris l'intérêt principal du multithreading, qui peut donner à un programme OO une apparente simultanéité aux agissements de plusieurs objets. Si ces objets tournent sur des processeurs différents, leur comportement sera, effectivement, parfaitement simultané. Dans la réalité, les objets se comportent de la sorte ; l'OO, qui a pour vocation d'y référer au mieux, ne pouvait déroger à cette nécessaire prise en compte.

Exercices

Exercice 17.1

Tentez de prédire ce qu'affichera le code Java suivant. Expliquez ce qu'il faudrait rajouter dans le code pour que l'alternance entre les deux threads soit parfaite :

```
public class ExempleThread extends Thread {
  static int j = 1;

  public void run()  {
    for (int i = 1; i<=200; i++) {
        System.out.println(j++ + " " + getName());
    }
  }

  public static void main(String[] args)  {
      new ExempleThread().start();
      new ExempleThread().start();
  }
}
```

Exercice 17.2

Expliquez ce qu'il faudrait rajouter dans ce code Java pour qu'à la sortie de l'imprimante, les pages des deux documents ne soient plus mélangées.

```
public class ExempleThread2 extends Thread {
  static Imprimante imprimante;
  String[] pages;
  public Exemple16(String s) { super(s);}
  public void run() {
    pages = new String[100];
    for (int i=0; i < pages.length; i++) {
        pages[i] = "Page " + (i+1) + " du " + getName();
    }
    imprimante.imprime(pages);
  }
  public static void main(String[] args) {
          imprimante = new Imprimante();
          new ExempleThread2("premierDocument").start();
           new ExempleThread2("deuxiemeDocument").start();
  }
}

class Imprimante {
  public void imprime(String[] s) {
    for (int i = 0; i<s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

Exercice 17.3

À partir du code C# ci-après, introduisez une troisième classe d'instruments dont les objets puissent jouer en même temps que les autres :

```
using System;
using System.Threading;

abstract class Instrument {
  private Thread unThread;

  public Instrument() {
     unThread = new Thread(new ThreadStart(jeJoue));
  }

  public Thread getThread() {
    return unThread;
  }

  abstract public void jeJoue();
}
```

```
class Guitare : Instrument {
  public Guitare(): base() {}

  public override void jeJoue() {
    for (int i = 0; i< 1000; i++) {
        Console.WriteLine(" je fais djing djing ");
    }
  }
}

class Trompette : Instrument {
  public Trompette() : base() {}
  public override void jeJoue() {
    for(int i = 0; i<1000; i++) {
        Console.WriteLine(" je fais pouet pouet ");
    }
  }
}

public class Orchestre {
  public static void Main() {
    Guitare uneGuitare = new Guitare();
    Trompette uneTrompette = new Trompette();
    uneGuitare.getThread().Start();
    uneTrompette.getThread().Start();
  }
}
```

Exercice 17.4

Réalisez, en Java ou en C#, un programme qui, grâce au multithreading, permette à plusieurs valeurs d'action boursière d'afficher leur fluctuation journalière en même temps sur l'écran.

Exercice 17.5

Pourquoi le multithreading est-il indispensable aux applications informatiques qui permettent l'achat d'article sur Internet ?

Exercice 17.6

Pourquoi la méthode `sleep()` est-elle statique ?

Exercice 17.7

Pourquoi faut-il synchroniser l'accès aux bases de données ?

Exercice 17.8

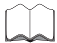
Pourquoi le résultat du multithreading peut dépendre de la plate-forme sur laquelle le programme s'exécute ?

18

Programmation événementielle

Ce chapitre décrit un mode plus implicite de communication entre les objets, où chacun d'eux peut être observé par un autre, de telle manière que la modification de l'état de l'objet observé déclenche l'exécution d'une méthode chez l'objet observant, tout cela se passant sans le moindre envoi de message.

Sommaire : Programmation événementielle — Interaction entre objets sans message explicite — En Java : classe Observable et interface Observer — En C# : les délégués et les « event » — Indirectement en Python



Candidus — Toutes ces idées nouvelles me font penser à la révolution logicielle qu'a représentée l'organisation événementielle des programmes X Window. Le jeu consistait à mettre en place toute une collection de « callbacks » permettant à l'utilisateur de déclencher le traitement de ses interventions.

Doctus — Il s'agissait effectivement d'une nouvelle façon de structurer le déroulement de nos programmes. Une interface graphique est le meilleur exemple qu'on puisse imaginer comme tableau de bord pour un système complexe.

Cand. — Justement, j'y pensais. Nos objets sont-ils prêts à nous permettre ce type d'accès vers leurs fonctionnalités ?

Doc. — Tu penses bien que oui ! L'OO est en fait à l'origine de la programmation événementielle des interfaces graphiques que sont les fenêtres, boutons et zones de texte.

Cand. — Ils sont donc prêts à réagir aux événements ! Comment vont-ils nous permettre d'établir les liaisons événements-méthodes ?

Doc. — D'abord une mise au point : cette liaison événement-action de nos premiers objets graphiques était assez sommaire. Elle consistait par exemple à associer un clic de la souris, effectué lorsque le pointeur était situé dans une certaine zone de l'écran, à l'invocation de certaines fonctions appelées « callbacks ». Ces fonctions étaient en quelque sorte des méthodes associées aux fenêtres du tableau de bord.

Cand. — Nous avions donc déjà des objets à part entière sans le savoir !

Doc. — Exactement ! Il s'agit maintenant de faire un pas de plus. Nous voulons traiter toutes sortes d'événements et ne plus nous limiter aux interfaces graphiques. Tous nos objets doivent pouvoir participer à un mécanisme de diffusion de l'information afin de pouvoir y mettre leur grain de sel, quand ils ont un rôle à jouer dans l'histoire.

Cand. — Il me semble qu'il leur suffit d'écouter la radio et de décrocher le téléphone pour intervenir !

Doc. — Y'a pas de radio dans la plupart de nos programmes. On peut en revanche prévoir un système d'abonnement aux seules informations utiles à chacun. Pour un objet particulier, les informations utiles sont en général assez peu nombreuses pour que ce type de liaison soit suffisant.

Cand. — Comment décrirais-tu le mécanisme qui est alors mis en place ?

Doc. — Nos sources d'information devront mémoriser la liste des objets qui s'intéressent à elles et penser à les joindre à chaque fois qu'elles changent d'état. De leur côté, les objets abonnés devront fournir un moyen d'être appelés par ces sources.

Cand. — Et je suppose qu'une étape de mise en place des interlocuteurs s'occupe de présenter les uns aux autres… joli travail !

Des objets qui s'observent

Nous savons, à ce stade du livre, que l'interaction entre les objets est la base de l'OO. Cette interaction, jusqu'ici, n'a été conçue que par envoi explicite de messages entre deux objets, quand le premier déclenche, « en pleine connaissance de cause », l'exécution d'une méthode sur le second. Le message peut être synchrone, interrompant alors la série des choses à faire par l'expéditeur, jusqu'à ce que l'exécution du message soit terminée par le destinataire. Le message peut, en revanche, être asynchrone, quand l'expéditeur et le destinataire, la communication à peine entamée, s'ignorent de plus belle, chacun affairé à ses propres instructions. Ce mécanisme peut avoir ceci de contraignant qu'il exige, dès la conception du code, de savoir à quel moment et pour quelle raison les messages seront envoyés, ainsi que de savoir à qui ils seront envoyés. Chaque objet expéditeur doit savoir à qui et comment il envoie son message, que ce soit au moment de la compilation ou lors de l'exécution.

Il existe pourtant une autre manière de penser la communication entre objets, relevant de la « programmation événementielle », une manière plus implicite, n'utilisant pas les envois de messages, mais exigeant que les objets s'observent plutôt qu'ils ne se parlent. Il s'agit de programmer en « couvrant les derniers événements ». Cette observation d'un objet par un autre permettra à l'observateur d'ajuster son tir en fonction de l'observé. On crée un lien de dépendance entre les objets plutôt que de communication. Ce mode de programmation sera appelé événementiel, car une chose qui se produit dans un objet sera considéré comme un événement pour un ensemble d'autres, qui devront alors agir en conséquence.

Cette dépendance entre les objets ne pourra être concrétisée dans aucune des deux classes concernées, l'observable et l'observateur, mais plutôt dans une troisième classe, qui établira ce lien de dépendance. Ainsi, lors de la Seconde Guerre mondiale, quand fut communiqué à la radio le fameux message « les violons de l'automne blessent mon cœur… », annonçant l'imminence du débarquement allié aux résistants et leur mise à contribution pour faciliter ce débarquement, l'auteur de ce message ignorait tout des auditeurs. Ces derniers attendaient sa diffusion, l'oreille collée au poste, pour agir en conséquence, sans que l'auteur n'ait la moindre connaissance de qui ils étaient, souhaitant toutefois qu'ils préfèrent Verlaine à « Lili Marlène ».

Un autre exemple, moins romantique mais qui se réfère directement au premier chapitre de ce livre, est celui du feu rouge et des voitures qui font rugir leur moteur en attendant de pouvoir démarrer en trombe dès que le feu vert s'affichera. Lorsque le feu indique aux conducteurs qu'ils peuvent enfin relâcher le frein et passer la première, il n'a aucune idée des interlocuteurs motorisés à qui il envoie ce message. Il serait plus naturel de concevoir un programme dans lequel les voitures devant être informées s'inscrivent comme « observateurs » du feu au fur et à mesure de leur arrivée devant celui-ci comme vous le verrez à la fin du chapitre

Bien sûr, le mode de programmation qui sous-tend en arrière-plan cette interaction reste l'envoi de messages, mais cette couche supplémentaire, réalisée comme un des vingt-trois « design patterns » auxquels nous consacrons

notre dernier chapitre, permet de penser l'interaction d'une manière différente et moins contraignante. La pratique demande très simplement, d'abord, d'une classe, qu'elle se transforme en une classe observable. Ce faisant, dans la déclaration de cette classe devra figurer, à un endroit précis de son code, son souci d'avertir tous ceux et celles qui l'observent, que s'est produit ce dont elles voulaient être informées. Ces dernières, ensuite, réagiront en conséquence. Dans leur code, effectivement, les classes qui observent doivent prévoir ce qu'elles feront au moment de l'occurrence de l'événement. Finalement, une troisième classe sera responsable de la relation s'établissant entre les observateurs et les observés, en inscrivant ceux-ci auprès de ceux-là.

Nous considérerons donc que, dans notre écosystème, situation parfaitement invraisemblable, la plante est observée par les prédateurs qui, dès que la plante est attaquée par la proie, en sont informés, de manière à se rabattre sur la proie, plutôt que sur la plante elle-même. Le code Java réalisant cela est présenté ci-après.

En Java

Tout d'abord la plante

```
import java.util.*;
public class Plante extends Observable { /* implémente la classe Observable */
  private int quantite;
  private boolean jeSuisConsomme;

  Plante (int quantite){
    this.quantite = quantite;
    jeSuisConsomme = false;
  }
  public boolean suisJeConsomme() {
    return jeSuisConsomme;
  }
  public void changeConsomme() {
    jeSuisConsomme = true;
    setChanged();  /* on signale l'état de l'objet à communiquer */
    notifyObservers(); /* on avertit qui de droit */
  }
  public void diminue (int decroit) {
    if (quantite > decroit) {
      System.out.println("ok, la plante diminue");
      quantite -= decroit;
    }
    else {
      quantite = 0;
      System.out.println("zut, il n'y a plus de plante");
    }
  }
}
```

La plante s'est transformée, par le mécanisme d'héritage, en une classe « observable » (qui est définie dans le package java.util). En tant que classe observable, elle doit, au moment où elle désire prévenir ceux qui l'observent, appeler `setChanged()`, pour signaler que c'est l'état de l'objet à cet instant qu'elle désire communiquer, et `notifyObserver()` qu'elle décide, effectivement, d'en aviser ses observateurs (comme vous pouvez le constater, elle n'a pas connaissance directe de qui ils sont). Ici, la plante veut simplement avertir ceux que cela

intéresse qu'elle est en train d'être consommée (et on répète qu'au contraire de l'envoi de message, elle n'a cure de savoir qui ils sont).

Du côté du prédateur

```
import java.util.*;

public class Predateur extends Thread implements Observer {
  private Eau uneEau;
  private Plante unePlante;
  private boolean faimPlante;

  Predateur (Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
    faimPlante = true;
  }
  public void jeBois() {
    for (int i=0; i<100; i++) {
      System.out.println("le predateur essaie de boire");
      uneEau.diminue(20);
    }
  }
  public void update(Observable o, Object arg) {
    /* hérité de l'interface et à redéfinir absolument, indique ce qui sera fait suite à l'événement */
    faimPlante = false;
    System.out.println("moi, predateur, je n'ai plus faim de plante mais de proie");
  }
  public void jeMange() {
    if (faimPlante) {
      synchronized (unePlante) {
        for (int i=0; i<25; i++) {
          System.out.println("le predateur essaie de manger");
          unePlante.diminue(20);
        }
      }
    }
    else {
      System.out.println("miam-miam, à nous deux la proie");
    }
  }
  public void run() {
    jeBois();
    jeMange();
  }
}
```

Le prédateur, lui, se doit d'implémenter l'interface `Observer`, afin de déclarer la manière dont il souhaite réagir à la transformation d'état de l'objet observé. Il le fait, simplement (mais de façon obligatoire car elle déclare implémenter l'interface `Observer`), en redéfinissant la méthode `update (Observable o, Object arg)`. Cette méthode peut recevoir, *via* ses arguments, des informations sur l'objet observable que la méthode `notifyObserver` pourra lui transmettre. Nous verrons pratiquement dans l'exemple du `FeuDeSignalisation` comme cela se passe. Ici, banalement, le prédateur, apprenant que la plante est consommée, décide de changer de cible et de s'attaquer à la proie.

Du côté de la proie

```
public class Proie extends Thread {
  private Eau uneEau;
  private Plante unePlante;

  Proie(Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
  }
  public void jeBois() {
    for (int i=0; i<50; i++) {
      System.out.println("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
  public void jeMange() {
    unePlante.changeConsomme();
    synchronized (unePlante) {
      for (int i=0; i<25; i++) {
        System.out.println("la proie essaie de manger");
        unePlante.diminue(10);
      }
    }
  }
  public void run() {
    jeBois();
    jeMange();
  }
}
```

C'est la proie qui, en attaquant la plante, avise celle-ci qu'elle est consommée.

Finalement, du côté de la Jungle

```
public class Jungle {
  public static void main(String[] args) {
    Eau uneEau              = new Eau(1000);
    Plante unePlante        = new Plante(1000);
    Proie uneProie          = new Proie(uneEau, unePlante);
    Predateur unPredateur   = new Predateur(uneEau, unePlante);
    uneProie.start();
    unPredateur.start();
    unePlante.addObserver(unPredateur); /* Le prédateur devient un observateur de la proie */
  }
}
```

La jungle fait du prédateur un nouvel observateur privilégié de la proie. Le prédateur s'est abonné au nouvel observateur... Par la méthode `addObserver()`, c'est la jungle qui installe le lien de dépendance entre les deux objets, et permettra au prédateur d'être avisé des changements d'état qui l'intéressent dans la plante. Le résultat de l'exécution est indiqué ci-après :

Résultat

```
…
le prédateur essaie de boire
la proie essaie de boire
OK ! l’eau diminue
OK ! l’eau diminue
le prédateur essaie de boire
la proie essaie de boire
OK ! l’eau diminue
OK ! l’eau diminue
moi, prédateur, je n’ai plus faim de plante mais de proie
la proie essaie de manger
OK ! la plante diminue
la proie essaie de manger
OK ! la plante diminue
la proie essaie de manger
…
zut ! il n’y a plus d’eau
le prédateur essaie de boire
zut ! il n’y a plus d’eau
le prédateur essaie de boire
zut ! il n’y a plus d’eau
…
zut ! il n’y a plus d’eau
miam-miam, à nous deux la proie
```

Le prédateur ayant été avisé de la consommation de la plante par la proie (on le constate lors de l’écriture de la ligne « moi, prédateur, je n’ai plus faim de plante mais de proie »), lorsqu’il décide de manger, il choisit de s’attaquer directement à la proie plutôt qu’à la plante. On le comprend.

La version C# de ce même mécanisme de dépendance événementielle est assez différente, car elle met en pratique cette originalité syntaxique et très efficace de C# (et .Net en général) que sont les délégués, ces référents de méthode.

En C# : les délégués

Généralités sur les délégués dans .Net

Dans .Net, les délégués sont des référents de méthode qui permettent de séparer dans l’écriture du code l’appel d’une méthode de son implémentation. La définition de la méthode peut être différée par rapport à l’appel de celle-ci à même le code. La déclaration du délégué sera conforme à la signature de toutes les méthodes cherchant à se substituer à celui-ci lors de l’exécution. Le nom « délégué » exprime la possibilité pour ceux-ci d’être « délégués » par les véritables méthodes, le temps que celles-ci soient définies. Cela pourrait ressembler à des interfaces, mais limitées à une seule méthode. Les clients ne connaissent que l’interface lors de la compilation, et la méthode ne sera trouvée et mise en œuvre qu’au moment de l’exécution. On fait tout avec ce délégué, sauf définir son bloc d’instructions, et son association avec la méthode qui le concrétise précisément et qui se produit au moment de l’exécution. Le petit code C# qui suit devrait clarifier ce mécanisme :

```
using System;

public class TestDelegue {
      public delegate void exempleDelegue(String message); // déclaration du délégué
```

```
        public static void Main() {
                Testeur t = new Testeur();
                exempleDelegue ed = new exempleDelegue(t.faire); // association à la méthode
               ed("Bersini"); // exécution de la méthode
        }
}

class Testeur {
        public void faire(String message) { // une concrétisation possible du délégué
                        Console.WriteLine("Bonjour " + message);
        }
}
```

Résultat

```
Bonjour Bersini
```

Dans ce code, le délégué est d'abord déclaré comme ne pouvant s'associer qu'à des méthodes ne renvoyant rien mais recevant un `String` en paramètre. Ensuite, une instance du délégué `ed` est créée, que l'on associe à la méthode `t.faire` afin qu'elle soit conforme à cette signature. On exécute ensuite le délégué comme s'il s'agissait de la méthode en lui passant l'argument en question. L'exemple suivant montre qu'un même délégué peut recevoir une suite de méthodes, toutes signées de la même façon. La première méthode est non statique et exige la création d'une instance de la classe `Testeur` ; la deuxième est statique et peut fonctionner directement à partir de la classe.

```
using System;

public class TestDelegue {
        public delegate void exempleDelegue(String message);

        public static void Main() {
                Testeur t = new Testeur();
                exempleDelegue ed = new exempleDelegue(t.faire);
                ed += new exempleDelegue(Testeur.faireAutrement); // ajout d'une nouvelle méthode
               ed("Bersini");
        }
}

class Testeur {
        public void faire(String message) {
                Console.WriteLine("Bonjour " + message);
        }

        public static void faireAutrement(String message) {
                Console.WriteLine("Au revoir " + message);
        }
}
```

Résultat

```
Bonjour Bersini
Au revoir Bersini
```

Nous avons vu ce mécanisme à l'œuvre lors de la mise en pratique du multithreading « à la sauce » .Net, dans le chapitre précédent. Pour ce faire, un délégué de type `ThreadStart` peut s'associer à n'importe quel corps de méthode renvoyant `void` et ne recevant aucun argument. Il s'agit bien du corps d'instructions associé au thread. On conçoit dès lors l'équivalence fonctionnelle des interfaces (à la Java) et des délégués (à la .Net). On retrouve cette même équivalence dans la mise en œuvre des délégués pour implémenter les fonctionnalités « bouton » et « souris » dans .Net, comme le petit code ci-dessous l'illustre : un bouton est créé puis disposé sur une fenêtre. Lorsqu'on on clique sur ce même bouton, ce qui se produit est défini par la méthode associée au délégué `EventHandler`.

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class BasicWindowsForm : Form {
       protected Button bouton;

public BasicWindowsForm()
{
      Init();
}

private void Init() {
      this.Size = new Size(400, 400);
      this.Text = "Basic Windows Form";
      CreeBouton();
}

private void CreeBouton() {
      bouton = new Button();
      bouton.Text = "Hello World";
      bouton.Dock = DockStyle.Fill;
      this.Controls.Add(bouton);

      bouton.Click += new EventHandler(Handler);  /* association entre le délégué et
              son corps, l'appel se fera lorsqu'on clique sur le bouton */
}

private void Handler(object envoyeur, EventArgs e) { // implémentation du délégué
       Console.WriteLine(e.ToString() + " " + envoyeur.ToString());
      Console.WriteLine("Hello World");
}

public static void Main() {
      Application.Run (new BasicWindowsForm());
}
}
```

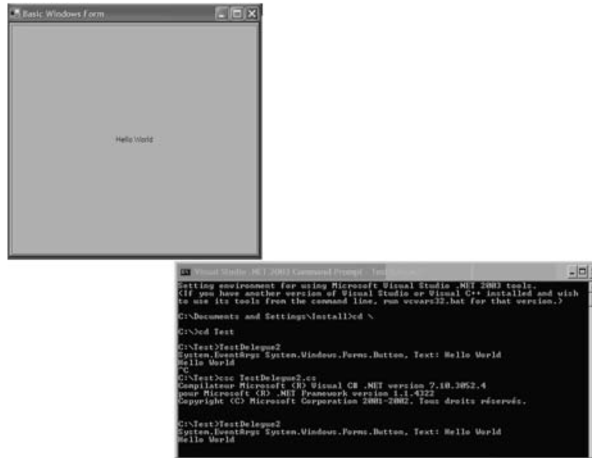


Figure 18-1

Illustration de l'utilisation du délégué EventHandler pour la gestion d'un bouton.

Retour aux observateurs et observables

Revenons à nos moutons, plus précisément à nos proies et prédateurs qui s'observent en .Net et reproduisons à l'aide des délégués l'équivalent du code réalisé précédemment en Java.

Tout d'abord, la plante

```
using System;
using System.Threading;

/* creation d'un nouveau type de délégué que l'on rattachera aux methodes à déclencher suite
   ➥à l'événement */
public delegate void GereObservation(object source);

class Plante {
  private int quantite;
  private Boolean jeSuisConsomme;
  /* on déclare ces délégués comme étant de type " event " car ils ne doivent
   * s'activer qu'en réponse à un événement */
  public event GereObservation observeurs;

  public Plante (int quantite) {
    this.quantite = quantite;
  }
  public Boolean suisJeConsomme() {
    return jeSuisConsomme;
  }
  public void changeConsomme() {
    jeSuisConsomme = true;
    /* on signale aux observateurs qu'il peuvent réagir en exécutant le délégué,
     * des informations sur l'objet peuvent être passées */
    observeurs(this);
```

```
    }
    public void diminue (int decroit) {
      if (quantite > decroit) {
        Console.WriteLine("ok, la plante diminue");
        quantite -= decroit;
      }
      else {
        quantite = 0;
        Console.WriteLine("zut, il n'y a plus de plante");
      }
    }
  }
```

Avant la définition de la classe `Plante`, une nouvelle catégorie de délégués est créée, appelée `GereObservation`. On sait, comme tout délégué en C#, que l'on créera à partir de ceux-ci des référents sur des méthodes. En fait, on indique précisément par ces délégués quelles méthodes doivent se déclencher dans la classe `Observateur`, quand l'observable change d'état. Pour ce faire, on rend ces délégués « événementiels » dans la plante, en les déclarant de type « event ». La plante associe au déclenchement d'un événement donné, ici, dès qu'elle est consommée, un appel à toutes les méthodes référées par le délégué. Elle le fait par l'instruction `observeurs(this)`, qui signalera aux observateurs qu'ils peuvent réagir en conséquence. Comme pour Java, il est prévu dans la définition du délégué que des informations sur la source de l'événement puissent être transmises.

Du côté du prédateur

```
  class Predateur {
    private Eau uneEau;
    private Thread unThread;
    private Plante unePlante;
    private Boolean faimPlante;

    public Predateur(Eau uneEau, Plante unePlante) {
      this.uneEau = uneEau;
      this.unePlante = unePlante;
      unThread = new Thread(new ThreadStart(jeConsomme));
      faimPlante = true;
    }
    public void lanceThread() {
      unThread.Start();
    }
    public void jeConsomme() {
      jeBois();
      jeMange();
    }
    public void jeBois() {
      for (int i=0; i<100; i++) {
        Console.WriteLine("le predateur essaie de boire");
        uneEau.diminue(20);
      }
    }
```

```
  public void miseAJour(object source) {
    faimPlante = false;
    Console.WriteLine("moi, predateur, j'ai plus faim de plante mais de proie");
  }
  public void jeMange() {
    if (faimPlante) {
      Monitor.Enter (unePlante);
      for (int i=0; i<100; i++) {
        Console.WriteLine("le predateur essaie de manger");
        unePlante.diminue(10);
      }
      Monitor.Exit (unePlante);
    }
    else {
      Console.WriteLine("miam-miam, à nous deux la proie");
    }
  }
}
```

Rien de spécial pour le prédateur, si ce n'est l'addition d'une méthode `miseAJour()` qui joue le même rôle que la méthode `update()` en Java. Ici, au contraire de Java, le nom de cette méthode importe peu mais, lors de la création de délégués, plus tard, cette méthode sera passée en référent, comme celle qui doit être exécutée en réponse à un événement donné.

Du côté de la proie

```
class Proie  {
  private Eau uneEau;
  private Thread unThread;
  private Plante unePlante;

  public Proie(Eau uneEau, Plante unePlante) {
    this.uneEau = uneEau;
    this.unePlante = unePlante;
    unThread = new Thread(new ThreadStart(jeConsomme));
  }
  public void jeConsomme() {
    jeBois();
    jeMange();
  }
  public void jeBois(){
    for (int i=0; i<50; i++) {
      Console.WriteLine("la proie essaie de boire");
      uneEau.diminue(10);
    }
  }
  public void jeMange(){
    unePlante.changeConsomme();
    Monitor.Enter (unePlante);
```

```
    for (int i=0; i<25; i++){
      Console.WriteLine("la proie essaie de manger");
      unePlante.diminue(10);
    }
    Monitor.Exit (unePlante);
  }
  public void lanceThread() {
    unThread.Start();
  }
}
```

C'est le même genre de proie que pour Java. Donc, rien de spécial de ce côté.

Finalement, du côté de la Jungle

```
public class Jungle{
  public static void Main(){
    Eau uneEau             = new Eau(1000);
    Plante unePlante       = new Plante(1000);
    Proie uneProie         = new Proie(uneEau, unePlante);
    Predateur unPredateur  = new Predateur(uneEau, unePlante);

    /* c'est maintenant que, d'abord, on associe le délégué à la méthode "miseAJour"
     * et, de plus, que l'on crée la dépendance entre la plante et le prédateur */
    unePlante.observeurs += new GereObservation (unPredateur.miseAJour);
    uneProie.lanceThread();
    unPredateur.lanceThread();
  }
}
```

Résultat

```
comme pour le code Java
```

Comme en Java, c'est dans la `Jungle` que va s'installer la dépendance, ici, non pas entre l'observable et les observateurs, mais plus directement entre l'événement qui se produit dans l'observable et les méthodes à déclencher en réponse à cet événement. Le délégué `GereObservation` référera maintenant la méthode `miseAJour` du prédateur. De surcroît, c'est cette mise à jour qui se déclenchera en réponse à l'événement. Toutes ces méthodes seront référencées par les `event GereObservation observeurs` de plante. Ici, une seule est concernée, la méthode `miseAJour` du prédateur, mais d'autres pourraient l'être, comme le montre bien la présence du +=.

Voici un autre exemple illustrant comment des objets `dormeur` peuvent s'inscrire comme autant d'observeurs d'un objet `réveil`. Marcel et Maurice associent leur méthode `seReveiller` au réveil. Maurice y rajoute par la suite sa méthode `continueADormir`. Chaque dormeur, ainsi que le réveil, possède son propre thread. Dès que le réveil sonne, les deux threads des dormeurs sont interrompus.

```
using System;
using System.Threading;

public class Test {
```

```
    public static void Main(String[] args) {
        Reveil reveil = new Reveil();
        Dormeur marcel = new Dormeur("Marcel");
        reveil.client += new trop (marcel.seReveiller);
        marcel.seCoucher(20);
        Dormeur maurice = new Dormeur("Maurice");
        reveil.client += new trop(maurice.seReveiller);

        reveil.client += new trop(maurice.continueADormir);


        maurice.seCoucher(20);
        reveil.demarre(100);
    }
}

public delegate void trop();

class Reveil {
    private int temps;
    public event trop client;

    public void demarre(int temps) {
        this.temps = temps;
        Thread threadReveil = new Thread(new ThreadStart(tictac));
        threadReveil.Start();
    }

    public void tictac() {
        for (int i=0; i < temps; i++) {
            Console.Write("tic...tac...");
            Thread.Sleep(200);
        }
        Console.WriteLine();
        Console.WriteLine("Drrrriiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
        iiiiiinnnnnnnnnnnnnnnnnnnnnng  !!!!");
        client();
    }
}

class Dormeur {
    Thread threadDormeur;
    String nom;
    public Dormeur(string nom){
        this.nom = nom;
    }
    public void seCoucher(int temps){
        Console.WriteLine("Bonjour, je m'appelle " + nom + " et je suis fatigué");
        Console.WriteLine("Je vais dormir " + temps + " minutes");
        threadDormeur = new Thread(new ThreadStart(dormir));
        threadDormeur.Start();
```

```
        Console.WriteLine(nom + "  est maintenant endormi   !");
    }
    public void dormir(){
        while (true){
            Console.Write("RRRR.....PSSSS......");
            Thread.Sleep(1000);}
    }
    public void seReveiller(){
        threadDormeur.Abort();
        Console.WriteLine("Bonjour, je m'appelle " + nom + " et je suis réveillé !");
    }

    public void continueADormir(){
        Console.WriteLine("Aujourd'hui, c'est dimanche. Je reste couché !");
    }
}
```

Résultat

```
Bonjour, je m'appelle Marcel et je suis fatigué
Je vais dormir 20 minutes
Marcel  est maintenant endormi   !
Bonjour, je m'appelle Maurice et je suis fatigué
Je vais dormir 20 minutes
Maurice  est maintenant endormi   !
c...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......RRRR.....PSSSS......
tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......
RRRR.....PSSSS......tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...
RRRR.....PSSSS......RRRR.....PSSSS......tic...tac...tic...tac...tic...tac...tic.
..tac...tic...tac...RRRR.....PSSSS......RRRR.....PSSSS......tic...tac...tic...ta
c...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......RRRR.....PSSSS......
tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......
RRRR.....PSSSS......tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...
RRRR.....PSSSS......RRRR.....PSSSS......tic...tac...tic...tac...tic...tac...tic.
..tac...tic...tac...RRRR.....PSSSS......RRRR.....PSSSS......tic...tac...tic...ta
c...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......RRRR.....PSSSS......
tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...RRRR.....PSSSS......
RRRR.....PSSSS......tic...tac...tic...tac...tic...tac...tic...tac...tic...tac...
RRRR.....PSSSS......RRRR.....PSSSS......
Drrrrriiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
nnnnnnnnnnnnnnnnnnnnnnng  !!!!
Bonjour, je m'appelle Marcel et je suis réveillé !
Bonjour, je m'appelle Maurice et je suis réveillé !
Aujourd'hui, c'est dimanche. Je reste couché !
```

En Python : tout reste à faire

Conformément à sa volonté de se limiter à l'essentiel, il n'existe pas à proprement parler de mécanisme prémâché en Python (comme les classes `Observable` et `Observer en Java`) permettant d'implémenter implicitement une programmation de type événementiel. Un des principes fondateurs de Python est que

« l'explicite » doit toujours être privilégié par rapport à « l'implicite ». Les mécanismes mis en œuvre dans le code Python ci-dessous permettent la reproduction des méthodes `setChanged`, `addObserver` et `update` qui caractérisent la solution Java.

```
import threading
import time
from threading import Thread

class Eau:
      __quantite=0
      def __init__(self,quantite):
              self.__quantite=quantite
      def diminue(self,decroit):
              if self.__quantite>decroit:
                        print "ok, l'eau diminue"
                        self.__quantite-=decroit
              else:
                        self.__quantite=0
                        print "zut, il n'y a plus d'eau"

class Plante:
      __quantite=0
      __jeSuisConsomme=0
      __observer=None
      __lock=0
      def __init__(self,quantite):
              self.__lock=threading.Semaphore(1)
              self.__quantite=quantite
      def suisJeConsomme(self):
              return self.__jeSuisConsomme
      def changeConsomme(self):
              self.__jeSuisConsomme=1
              self.setChanged()
      def diminue(self,decroit):
              self.__lock.acquire()
              if self.__quantite>decroit:
                        print "ok, la plante diminue"
                        self.__quantite-=decroit
              else:
                        self.__quantite=0
                        print "zut, il n'y a plus de plante"
              self.__lock.release()
      def setChanged(self) :
              self.__observer.update()
      def addObserver(self,observer):
              self.__observer=observer

class Predateur(Thread):
      __uneEau=None
      __unePlante=None
      __faimPlante=True
```

```
    def __init__(self,uneEau,unePlante):
        self.__uneEau=uneEau
        self.__unePlante=unePlante
        Thread.__init__(self)
    def jeBois(self):
        i=0
        while i<100:
            print "le predateur essaie de boire"
            self.__uneEau.diminue(20)
            i+=1
    def update(self):
        self.__faimPlante=False
            print "moi, predateur, je n'ai plus faim de plante mais de proie"
    def jeMange(self):
        if self.__faimPlante==True:
            i=0
            while i<25:
                print "le predateur essaie de manger"
                self.__unePlante.diminue(20)
                i+=1
        else:
            print "miam-miam, a nous deux la proie"
    def run(self):
        self.jeBois()
        self.jeMange()

class Proie(Thread):
    __uneEau=0
    __unePlante=0
    def __init__(self,uneEau,unePlante):
        self.__uneEau=uneEau
        self.__unePlante=unePlante
        Thread.__init__(self)
    def jeBois(self):
        i=0
        while i<50:
            time.sleep(0.01)
            print "la proie essaie de boire"
            self.__uneEau.diminue(10)
            i+=1
    def jeMange(self):
        self.__unePlante.changeConsomme()
        i=0
        while i<25:
            print "la proie essaie de manger"
            self.__unePlante.diminue(10)
            i+=1
    def run(self):
        self.jeBois()
        self.jeMange()

uneEau=Eau(1000)
```

```
unePlante=Plante(1000)
uneProie=Proie(uneEau,unePlante)
unPredateur=Predateur(uneEau,unePlante)
uneProie.start()
unPredateur.start()
unePlante.addObserver(unPredateur)
```

Un feu de signalisation plus réaliste

Le code qui suit reprend les exemples des premiers chapitres traitant de l'interaction entre le feu de signalisation et des voitures, mais cette fois dans une version plus réaliste où le feu n'a nul besoin de connaître les voitures qui lui font face et qui démarrerons dès son passage au vert. Le feu se contente d'être observé par des voitures qui se rajoutent à lui, non plus directement à partir de sa classe, mais par l'entremise d'une tierce classe (ici la classe `Traffic` qui fait le raccord entre le feu et les voitures). Lorsque le feu change de couleur, il ne le signale pas aux voitures, mais celles-ci (qui lui furent associées dans la classe `Traffic`) en sont malgré tout avisées et agiront en conséquence. De surcroît, la méthode `notifyObservers(Object)` peut transmettre certaines informations concernant la nature de l'événement (par exemple, ici, la couleur du feu) qui seront récupérées par la méthode `update (Observable, Object)`. De façon beaucoup plus logique, les voitures se rajoutent au feu au fur et à mesure de leur arrivée, ici gérée par la classe `Traffic`. La version C# ou Python sera facilement obtenue en transposant à cet exemple les codes dans les deux langages présentés précédemment.

En Java

```
import java.util.*;
import java.awt.event.*;

class FeuDeSignalisation extends Observable {
   private int couleur;

   public FeuDeSignalisation(int couleur){
      this.couleur = couleur;
   }

   public void changeCouleur () {
      couleur++;
      if (couleur == 4) couleur = 1;
      if (couleur == 1){
            setChanged();
            notifyObservers("vert");
      }
      else
      {
          if (couleur == 3){
             setChanged();
             notifyObservers("rouge");
          }
      }
   }
}
```

```
class Voiture implements Observer {
   private int vitesse;

   public Voiture(int vitesse){
      this.vitesse = vitesse;
   }

   public void demarre() {
      vitesse = 50;
      System.out.println("je demarre");
   }

   public void update(Observable o, Object arg) {
      System.out.println(o + " " + (String)arg);
      if ((String)arg == "vert") {
         demarre();
      }
   }
}

public class Traffic {
   public static void main(String[] args){
      FeuDeSignalisation unFeu = new FeuDeSignalisation(1);
      ArrayList<Voiture> l = new ArrayList<Voiture>();
      for (int i=0; i < 3; i++){
         l.add(new Voiture(0));
         unFeu.addObserver(l.get(i));
      }
      for (int i=0; i<4; i++){
         unFeu.changeCouleur();
      }
   }
}
```

Résultat

```
FeuDeSignalisation@5224ee rouge
FeuDeSignalisation@5224ee rouge
FeuDeSignalisation@5224ee rouge
FeuDeSignalisation@5224ee vert
je demarre
FeuDeSignalisation@5224ee vert
je demarre
FeuDeSignalisation@5224ee vert
je demarre
```

Deitel père et fils

Le livre que vous tenez entre les mains n'a pas pour vocation de faire de vous des experts dans chacune des technologies précises que nous découvrons au fur et à mesure des chapitres : C++, Java, C#, UML, RMI, Corba, services Web, bases de données relationnelles et OO, etc. L'ampleur de la tâche la rend tout simplement impensable. En revanche, nous espérons, en traversant l'un après l'autre les territoires technologiques évoqués, vous amener à la maîtrise de la pensée et la démarche orientée objet. Comme à travers le hublot d'un avion, vous aurez une meilleure perspective globale de l'OO, en survolant tous ces territoires, sans atterrir dans aucun d'entre eux. Nous avons opté pour cette traversée plutôt que la découverte en profondeur d'aucun des territoires, car c'est cette perspective plus globale et abstraite qui transcendera les années, alors que les technologies pointues, elles, ne font bien souvent que passer.

Si, maintenant, vous êtes à la recherche de guides de voyage pour chacun des territoires traversés, il est difficile aujourd'hui de ne pas vous recommander les best-sellers en la matière, les guides du routard de la planète OO, les livres des Deitel père et fils, dans la collection Prentice Hall. Ces livres ont pour sujet toutes les technologies que nous avons visitées : C, C++, C#, Internet, VB et VB.NET, XML avec en plus Perl, Python, et d'autres encore. Ils sont facilement repérables dans toutes les librairies, grâce à leur couleur criarde et leur design amusant, truffé de fourmis folles. Pour chacun de ces chapitres de l'OO, ils sont une référence de choix. Ils sont souvent très bien faits, assez complets, remplis d'exemples, et pleins d'esprit. Plus de 5000 universités se servent de leurs ouvrages pour les cours d'informatique, et ces mêmes ouvrages sont traduits dans toutes les langues des pays les plus développés (plusieurs le sont en français). Cela prouve, au contraire de ce que nous écrivions en introduction du chapitre 12, qu'il est des filiations et des reprises de flambeau familial qui se passent à merveille (y compris en informatique). Les Deitel et Deitel sont vraiment devenus une entreprise à part entière, ciblant tous les systèmes de formation informatique, et une petite visite sur leur site Web (*www.deitel.com*) vous en convaincra.

Exercices

Exercice 18.1

Différenciez la communication entre objets par envoi de messages de celle de type événementiel.

Exercice 18.2

Expliquez pourquoi la programmation événementielle permet à un objet de penser sa « stratégie de communication », sans pour autant définir à l'avance avec qui il communiquera.

Exercice 18.3

Réalisez, soit en Java, soit en C#, les deux classes suivantes, et pensez leur interaction de façon « événementielle ». Une première classe appelée `Loterie`, dont la seule méthode importante est celle qui consiste à générer des nombres de manière aléatoire. Une seconde classe appelée `Joueur`, observant la première, et dont la seule méthode sera de hurler sa joie, quand le numéro en sa possession sera généré par la classe `Loterie`.

Exercice 18.4

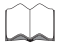
Réalisez en Java l'interaction entre une classe `Dormeur` et une classe `Réveil` de manière « événementielle », code réalisé en C# dans le chapitre. Récupérez par les dormeurs l'heure du réveil au moment précis où il sonne et, selon cette heure, choisissez de vous réveiller ou de retourner vous coucher.

19

Persistance d'objets

Dans ce chapitre, nous nous intéresserons aux différentes manières de sauvegarder, durant l'exécution du programme, les objets sur le disque dur.

Sommaire : Le problème de la persistance — Streams et sauvegarde en fichier — La sérialisation — Sauvegarde dans les bases de données — La correspondance entre diagramme de classes et schéma relationnel — Les bases de données orientées objet — La bibliothèque Linq de Microsoft



Candidus — Dis donc, je pense à quelque chose de dramatique : imaginons qu'un programme objet représente l'ensemble des instruments du tableau de bord d'un avion. Tout à coup, un orage survient, un éclair fait tout planter en une fraction de seconde. Comment faire pour limiter les dégâts ? Ne pourrait-on pas de temps en temps sauvegarder l'état de nos objets ailleurs que dans la mémoire RAM ?

Doctus — Ton exemple est un peu scabreux, mais il est néanmoins nécessaire de penser à enregistrer l'état de nos objets. Je suis sûr que tu en sais déjà assez pour décrire ce que pourrait comporter un tel mécanisme.

Cand. — Bon ! j'essaie. Une instance d'objet peut être identifiée par la classe à laquelle elle appartient. Ça nous donne déjà un identificateur auquel nous pourrions associer un fichier ou une entrée de table pour son enregistrement.

Doc. — Là, tu tiens déjà quelque chose. Le contenu de ta sauvegarde pourra effectivement donner lieu à une réinstanciation correcte.

Cand. — J'ai donc un moyen de raccrocher ma sauvegarde à l'ensemble des méthodes qui permettent d'en faire un nouvel exemplaire d'objet. Je suppose que la sauvegarde de ces méthodes est déjà assurée par le code exécutable ?

Doc. — C'est exact. Et que mettras-tu dans ta sauvegarde ?

Cand. — C'est simple, j'y mettrai tous les attributs de l'objet, autrement dit toutes les valeurs qui leur sont associées au moment de l'enregistrement.

Doc. — C'est bon pour les attributs de type primitif mais que feras-tu de ceux qui ont un type objet ?

Cand. — Ah oui ! Si je sauvegarde une simple référence, ce ne sera qu'une simple adresse mémoire. Au moment de la restauration, elle pointera sur des données qui risquent de ne pas contenir la même chose…

Doc. — Étends donc ton raisonnement de manière récursive et tu obtiendras ta solution.

Cand. — Tu veux dire que chaque attribut de type objet devra lui-même assurer sa sauvegarde ?

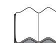
Doc. — C'est bien ça et chacun d'eux devra procéder de la même façon avec les siens. En fait, le meilleur moyen de les sauvegarder consiste à leur déléguer une partie du travail.

Cand. — Il faudra donc simplement implémenter des méthodes de sauvegarde dans chaque objet ?

Doc. — Même pas ! La plupart des objets procèdent tout bêtement du mécanisme que tu viens de décrire : sauvegarder les attributs primitifs, demander aux attributs de type objet de se sauvegarder eux-mêmes et mettre le tout dans un enregistrement qui pourra être restauré plus tard. Ce mécanisme standard est déjà réalisé en Java, C# et Python et PHP 5.

Cand. — Merci pour le « bêtement » ! Mais c'est vrai que nos objets sont des structures idéales pour le stockage en base de données.

Doc. — Malheureusement, ce n'est pas le cas ! Les structures OO ressemblent à s'y méprendre à celles des bases de données mais, quand il s'agit d'y sauvegarder les objets, un vrai travail de traduction doit être réalisé par le programmeur.



Sauvegarder l'état entre deux exécutions

Jusqu'à présent, lors de l'exécution d'un programme OO, des objets naissent, vivent en subissant un ensemble de transformations, puis meurent, broyés dans un sinistre camion-poubelle, à la fin du programme. Or, dans la grande majorité des applications, cette disparition des objets, emportant dans leur tombe toutes les transformations qu'ils ont subies, n'est pas des plus satisfaisantes. En effet, il est courant que ces transformations exigent d'être mémorisées pour que, lors d'une nouvelle exécution de ce programme, celui-ci puisse repartir de l'état des objets à la fin de l'itération précédente.

Pensez à des opérations bancaires, modifiant le solde de comptes en banque, ou à des jeux vidéo, dont vous désirez mémoriser l'état présent, avant de vaquer à d'autres occupations. Cela se passe exactement comme lors de l'utilisation du traitement de texte. Quand vous interrompez son utilisation, il vous demande si vous souhaitez sauvegarder toutes les modifications apportées à votre document (il est d'ailleurs plutôt conseillé dans notre monde informatique aujourd'hui, si stable, d'anticiper sa demande. Gardez toujours le petit doigt, non sur la couture du pantalon, mais sur ctr-s). Pour ce faire, il est capital de suppléer la grosse majorité des applications OO avec une mémoire persistante, en complément à la mémoire RAM (pile ou tas), qui n'a, elle, de rôle à jouer que pendant la seule exécution du programme.

Et que dure le disque dur

Aujourd'hui, dans vos ordinateurs, plusieurs systèmes de mémorisation transcendent le temps, les déclenchements et interruptions de programmes, la panne de courant, des cataclysmes, sans doute la fin du monde… Mais parmi ceux-ci, un seul dure, dure, dure… et durera encore longtemps, le disque dur. Ce disque engrange toutes les données, sous forme de blocs d'octets, encodés magnétiquement dans ses cylindres.

Le disque dur

Tout fichier est stocké physiquement comme un ensemble d'octets enchaînés par une liste liée (voir chapitre 21). Ces octets sont codés magnétiquement sur le disque. Ce dernier est doté d'une immense capacité, mais d'un accès très lent par rapport à la mémoire RAM. À chaque accès, il faut d'abord retrouver l'emplacement du bloc, le cylindre, le secteur, etc., puis commencer à le lire bit par bit et, quand il s'agit de l'entièreté du fichier, bloc par bloc. C'est un processus long et pénible, présentant le désavantage de ralentir les applications qui, au cours de leur déroulement, doivent extraire de l'information ou en déposer sur le disque dur.

Quatre manières d'assurer la persistance des objets

Si, pendant le déroulement de votre application, il vous paraît important de mémoriser les transformations opérées jusqu'à présent dans vos objets, il n'y a d'autres moyens de le faire que sur le disque dur. Nous allons, dans un esprit de continuité, supposer que, dans notre écosystème, et avant l'interruption du programme, nous souhaitions (exactement comme une partie de jeu vidéo) sauvegarder l'état des acteurs.

Il y a, aujourd'hui, quatre manières d'assurer la persistance des objets, qui, par sophistication croissante, sont : la simple écriture et lecture des valeurs d'attributs à sauvegarder sur un fichier, la sérialisation des objets, l'interaction avec une base de données relationnelle, et l'utilisation de bases de données orientées objet. Nous pourrions également les distinguer en fonction de leur respect de la nature « objet » des données à sauvegarder, leur facilité d'emploi, leur diffusion. Nous allons passer en revue ces quatre manières, leurs défauts, leurs qualités et l'état actuel de cette diffusion, en les appliquant à la sauvegarde de notre prédateur (ce qui ne devrait pas déplaire à la SPA).

Simple sauvegarde sur fichier

Utilisation des « streams »

Tous les langages de programmation dont nous nous servons permettent, par l'utilisation de « streams » (traduction littérale : « flux »), et de manière assez semblable, d'écrire et de lire des octets sur un fichier. Un « stream » est un tube à octet, qui connecte le programme en train de s'exécuter et tout type de périphérique, disque et autres (réseau, imprimantes…). Cette abstraction permet de traiter tous les périphériques comme un seul, en phase avec la manière dont le processeur se connecte aux différents périphériques : un bus unique, que la présence de « pilotes » permet ensuite de ramifier et de spécialiser en fonction du périphérique.

Comme à l'état primitif le « stream » ne fait circuler que des octets, cette classe est souvent sujette à des spécifications par le jeu de l'héritage, ou à des emboîtements (en Java et en C#) par lesquels on spécialise, par un second « stream », connecté à même le premier, un mode d'écriture ou de lecture sur le périphérique. On parle alors de « flux filtré ». Par exemple, les données à lire peuvent être des nombres, des caractères, des images, du son. Selon le cas, la manière d'associer les octets et de les transmettre à l'application est différente.

Par exemple, en Java, si vous désirez extraire un fichier des informations que vous savez, à l'avance, n'être que des nombres réels, vous allez emboîter sur un `FileInputStream` directement attaché au fichier en question (dans le code ci-après : le fichier appelé `fichier.dat`), un `DataInputStream`, et ce de la manière suivante :

```
FileInputStream fin = new FileInputStream ("fichier.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

La méthode `readDouble()` n'existe que dans la classe `DataInputStream`. Un autre type de filtrage très fréquent consiste à adjoindre, soit à la lecture, soit à l'écriture, un mécanisme de temporisation (en anglais « buffering »), qui permet, par exemple, d'écrire les octets dans une zone RAM locale, avant de transférer d'un seul coup tous ces octets sur le disque dur (pour minimiser le nombre d'accès disques, très lents). Ces flux filtrés sont un exemple de l'application du design pattern « décorateur » que nous découvrirons au chapitre 23.

L'héritage entre les « streams » est une caractéristique commune aux langages avec, comme toujours, au grand dam des programmeurs, quelques nuances de syntaxe ou de pratique, ici et là, les différenciant. La spécialisation de la lecture ou de l'écriture se fait souvent en fonction du type de périphérique avec lequel on communique, ou en fonction du type très particulier d'information à échanger. Comme nous le verrons dans la

suite, la lecture et l'écriture d'objets demanderont un « stream » dédié. On conçoit aisément la raison d'être de l'héritage. Dans tous les cas, on lit ou on écrit, mais il est nécessaire de redéfinir le mode de lecture ou le mode d'écriture en fonction du support ou des données à transmettre.

Qui sauve quoi ?

Effectuons, dans la suite, la sauvegarde de quatre données concernant chaque prédateur, son type, sa vitesse en x et en y, et son énergie, dans un fichier texte appelé `leFichierPredateur.txt`. Nous supposerons, de fait, que la lecture et l'écriture soient réalisées de la manière la plus simple qui soit, comme s'il s'agissait de lire, une après l'autre, les lignes de texte qui composent le fichier. Il est clair qu'il existe de nombreuses façons possibles de lire et écrire sur un fichier, comme il existe de nombreuses sous-classes `stream` à exploiter.

Nous nous bornerons ici, à une pratique fort simple, mais qui a l'insigne avantage de se transposer assez immédiatement d'un langage à l'autre. Nous supposerons, et c'est plutôt une bonne pratique, que le prédateur soit responsable de la persistance de ses données. Qui, mieux que lui, pourrait s'en occuper, puisque lui seul y accède de la manière la plus directe qui soit. Deux méthodes sont rajoutées dans la classe `Predateur` (mais elles pourraient l'être plus haut dans l'organisation hiérarchique des classes, car elles sont communes à d'autres classes et sous-classes), que nous nommons `sauveDonnees()`, responsable de l'écriture sur le disque, et `litLesDonnees()` responsable de la lecture. Ce sont ces deux méthodes dont le code sera récrit en fonction des mécanismes de persistance mis en œuvre. Nous n'indiquerons, pour chacun des langages, que les parties de code additionnelles, qui permettent l'interaction avec le fichier.

En Java

```
import java.io.*;
public class Predateur {
  public void sauveDonnees(PrintWriter maSortie) { /* écrit les données sur le fichier */
    maSortie.println(toString());
    maSortie.println(getVitX());
    maSortie.println(getVitY());
    maSortie.println(getEnergie());
    maSortie.println();
  }
  public void litLesDonnees(BufferedReader monEntree) { /* lit les données du fichier */
    try{
      String s=null;
      monEntree.readLine();  /* saute une ligne */
      s = monEntree.readLine(); /* lit une ligne */
      int vitx = Integer.parseInt(s); /* caste en entier */
      s = monEntree.readLine();
      int vity = Integer.parseInt(s);
      s = monEntree.readLine();
      double energie = Double.parseDouble(s); /* caste en double */
      setVitesse(vitx, vity);
      setEnergie(energie);
      monEntree.readLine(); /* saute le type */
    } catch (IOException e) {System.out.println(e); }
  }
}
```

```
public class Jungle {
  private FileOutputStream fos;     /* la connexion au fichier en écriture */
  private PrintWriter maSortie;     /* afin d'utiliser "println()" */
  private FileReader fr;            /* la connexion au fichier en lecture de texte */
  private BufferedReader monEntree; /* lecture "temporisée" */

  public Jungle(int largeur, int hauteur) {
    BufferedReader monEntree = null;
    try {
      fr = new FileReader("leFichierPredateur.txt"); /* connexion-fichier en lecture de texte */
      monEntree = new BufferedReader(fr);            /* flux filtré pour la lecture temporisée */
    } catch (IOException e) {e.getMessage();}
    for (int i=0; i<lesLions.length; i++)
      lesLions[i].litLesDonnees(monEntree);
    try{
      monEntree.close();  /* fermeture du tube */
    } catch (Exception e) {System.out.println(e);}
    try {
     fos = new FileOutputStream("leFichierPredateur.txt"); /* connexion sur le fichier en écriture */
    } catch (IOException e) {e.getMessage();}
    maSortie = new PrintWriter(fos, true);  /* flux filtré pour l'écriture de ligne de texte */
    for(int i=0; i<lesLions.length; i++)
      lesLions[i].sauveDonnees(maSortie);
    maSortie.close(); /* fermeture indispensable du tube */
  }
}
```

Nous commencerons par la classe `Jungle`. De manière quelque peu étrange (car ces deux opérations n'ont pas l'habitude de se suivre de si près), mais nous permettant d'illustrer tant l'écriture que la lecture, la jungle demandera d'abord à ses prédateurs de sauvegarder leurs données sur le disque, puis de les récupérer. Quatre déclarations de classe, deux pour la lecture et deux pour l'écriture, seront nécessaires. Pour la lecture, les deux classes en jeu sont `FileReader`, qui se limite à installer un « tube à caractères » entre le fichier et le programme, en signalant qu'il s'agit d'une lecture du fichier, mais d'une lecture spécialisée pour du texte, et `BufferedReader`, qui s'emboîte sur le premier, et met à disposition un mécanisme de temporisation qui rend la lecture plus efficace. La temporisation permet d'économiser les accès très coûteux en temps sur le disque dur. Pendant une large partie du programme, le tampon, zone de mémoire RAM, fait office de disque dur, jusqu'à ce qu'il soit « décharger » véritablement sur le disque dur.

Dans cette classe, la méthode `readLine()` lit un fichier ligne de texte par ligne de texte. Pour l'écriture, les deux classes en jeu sont : `FileOutputStream`, qui se limite à installer un « tube à octet » entre le fichier et le programme, en signalant qu'il s'agit d'un tube servant à l'écriture cette fois, et `PrintWriter`, qui spécialise cette lecture à du texte (et qui permettra, comme pour afficher du texte à l'écran, d'utiliser la méthode `println()`). Java, comme à l'habitude, force la gestion d'exception, pour gérer des imprévus, ici, principalement des problèmes d'accès sur le disque dur.

Dans la classe `Predateur`, la méthode `sauveDonnees()` se borne à écrire les quatre informations suivies d'une ligne vide de séparation. Elle écrit chaque information de manière non optimale (vu qu'il s'agit pour l'essentiel de nombre), comme une chaîne de caractères. Connaissant la nature des données, nous aurions pu recourir à l'utilisation de classes diverses pour réaliser tant la sauvegarde que l'écriture, les différentes options ne manquant pas en Java.

La méthode `litLesDonnees()` commence par lire chaque ligne du fichier, puis utilise un mode de conversion propre à Java, pour traduire ces données codées sur des chaînes de caractères, en entier ou double respectif. Finalement, il est toujours très important de fermer un tube d'accès fichier après l'usage, par la méthode `close()`, au risque que les données temporisées ne soient pas transmises comme il se doit, dans un sens comme dans l'autre. Cela permet également de libérer la connexion pour d'autres utilisations ou utilisateurs.

En C#

```
using System;
using System.IO;
class Predateur {
  public void sauveDonnees(StreamWriter maSortie) { /* écrit les données sur le fichier */
    maSortie.WriteLine(this);
    maSortie.WriteLine(vitX);
    maSortie.WriteLine(vitY);
    maSortie.WriteLine(energie);
    maSortie.WriteLine();
  }
  public void litLesDonnees(StreamReader monEntree) { /* lit les données du fichier */
    string s;
    monEntree.ReadLine();
    s = monEntree.ReadLine();
    int vitX =  Int32.Parse(s);
    s = monEntree.ReadLine();
    int vitY = Int32.Parse(s);
    s = monEntree.ReadLine();
    double energie = Double.Parse(s);
    setVitesse(vitX, vitY) ;
    setEnergie(energie) ;
    monEntree.ReadLine();
  }
}
public class Jungle {
  public static void Main() {
    /* obtient la connexion au fichier en mode lecture : */
    FileStream fso = new FileStream ("leFichierPredateur.txt", FileMode.Open);
    /* flux filtré pour utiliser "ReadLine()" : */
    StreamReader monEntree = new StreamReader(fso);
    for (int i=0; i<lesLions.length; i++)
      lesLions[i].litLesDonnees(monEntree);
    monEntree.Close(); /* fermeture du tube */
    /* obtient la connexion en mode écriture */
    FileStream fsc = new FileStream("leFichierPredateur.txt", FileMode.Create);
    /* flux filtré pour utiliser "WriteLine()" */
    StreamWriter maSortie = new StreamWriter(fsc);
    for(int i=0; i<lesLions.length; i++)
      lesLions[i].sauveDonnees(maSortie);
    maSortie.Close(); /* fermeture indispensable du tube */
  }
}
```

À quelques différences syntaxiques près, le code C# est très proche du code Java, en adoptant le même principe de flux filtrés. La différence entre l'écriture et la lecture se fait dans les arguments du constructeur du FileStream, et ne requiert pas de classes séparées. Ces arguments suffisent à spécifier le mode d'accès ou l'emploi d'un mécanisme de temporisation. Par exemple, le FileMode peut être `OpenOrCreate`, indiquant que le fichier doit être ouvert s'il existe, ou créé s'il n'existe pas. D'autres informations peuvent être spécifiées à la construction, comme `FileAccess.Read` quand seule la lecture est autorisée, ou `FileAccess.ReadWrite`, qui est l'option par défaut. Enfin, on retrouve, comme en Java, les mêmes « reader ou writer » pour la gestion de texte.

En C++

```
class Predateur {
  public:
    void sauveLesDonnees(ofstream maSortie) { /* écrit les données sur le fichier */
      maSortie << this << endl; /* surcharge de l'opérateur " << " */
      maSortie << vitX << endl;
      maSortie << vitY << endl;
      maSortie << energie << endl;
      maSortie << endl;
    }
    void litLesDonnees(ifstream monEntree) { /* lit les données du fichier */
      monEntree >> vitX; /* surcharge de l'opérateur ">>" */
      monEntree >> vitX;
      cout << vitX << endl;
      monEntree >> vitY;
      monEntree >> energie;
    }
};
int main(int argc, char* argv[]) {
  ifstream fis ("leFichierPredateur.txt"); /* obtient la connexion en mode lecture */
  if (!fis) {
    cerr << "leFichier n'est pas accessible" << endl;
    return false;
  }
  for (int i=0; i<lesLions.length; i++)
    lesLions[i]->litLesDonnees(monEntree);
  fis.close(); /* ferme le tube */
  ofstream fos ("leFichierPredateur.txt"); /* obtient la connexion en mode écriture */
  if (!fos) {
    cerr << "leFichier n'est pas accessible" << endl;
    return false;
  }
  for(int i=0; i<lesLions.length; i++)
    lesLions[i].sauveDonnees(maSortie);
  fos.close(); /* ferme le tube */
  return 0;
}
```

Comme à son habitude, la version C++ se distingue quelque peu. On y retrouve les flux d'entrée et de sortie, également installés dans une structure hiérarchique d'héritage. La gestion d'exception n'étant pas obligatoire, un test est effectué sur les pointeurs du fichier pour s'assurer que l'accès est possible. Ce qu'il y a de plus original dans cette version, c'est la ré-utilisation des mêmes << et >> pour l'écriture et la lecture, quoi qu'on écrive et où qu'on écrive (clavier/écran ou sur fichier). Cette ré-utilisation est possible, grâce à ce mécanisme très puissant de surcharge d'opérateur, qui permet, ici, de généraliser l'utilisation de << et > > comme il se doit, en fonction, à droite, du type de données transmises et, à gauche, du support sur lequel s'écrit ou se lit ces données.

En Python

```
class Predateur:
      def sauveDonnees(self,maSortie):
              try:
                    maSortie.write(self.__str__()+"\n")
                    maSortie.write(str(self.getVitX())+"\n")
                    maSortie.write(str(self.getVitY())+"\n")
                    maSortie.write(str(self.getEnergie())+"\n")
                    maSortie.write("fin du fichier\n")
              except:
                    print "une erreur s'est produite"
      def litLesDonnees(self,monEntree):
              try:
                    s=monEntree.readline()
                    s=monEntree.readline()
                    vitx=int(s)
                    s=monEntree.readline()
                    vity=int(s)
                    s=monEntree.readline()
                    energie=float(s)
                    self.setVitesse(vitx,vity)
                    self.setEnergie(energie)
                    monEntree.readline()
              except:
                    print "une erreur s'est produite"

uneEau=Eau(1000)
unePlante=Plante(1000)
uneProie=Proie(uneEau,unePlante)
unPredateur=Predateur(uneEau,unePlante)
try:
      fr=open('leFichierPredateur.txt', 'w')
      unPredateur.sauveDonnees(fr)
      fr.close()
      fr=open('leFichierPredateur.txt', 'r')
      unPredateur.litLesDonnees(fr)
      fr.close()
except:
      print "une erreur s'est produite"
```

Exactement comme en Java, on ouvre un fichier devenu `objet` pour l'occasion via la fonction `open` en lecture `r` ou en écriture `w` et on modifie son contenu via les fonctions `write` et `readline`. On referme ensuite le flux à l'aide de la méthode `close`. Remarquez que le troisième argument d'`open`, pour autant qu'on le spécifie, est la taille du buffer (attribuée par défaut par le système d'exploitation). Notez l'extrême simplicité du `casting` (toujours cette formidable brièveté d'écriture de Python) grâce aux méthodes `str`, `int` et `float`, ainsi que le mécanisme de levée d'exception qui remplace le `catch` par un `except`, mais n'oblige pas la récupération d'une exception à proprement parler (généralement, on spécifie malgré tout un type d'exception à récupérer).

En PHP 5

```
<html>
<head>
<title> Lecture et écriture fichier </title>
</head>
<body>
<h1> Lecture et écriture fichier </h1>
<br>

<?php
 class Predateur {
             private $vitX;
             private $vitY;
             private $energie;

             public function __construct($vitX, $vitY, $energie){
                          $this->vitX = $vitX;
                          $this->vitY = $vitY;
                          $this->energie = $energie;
             }

             public function sauveDonnees($maSortie){
                          fwrite($maSortie,"$this\n");
                          fwrite($maSortie,"$this->vitX\n");
                          fwrite($maSortie,"$this->vitY\n");
                          fwrite($maSortie,"$this->energie\n");
                          fwrite($maSortie,"\n");
             }

             public function litLesDonnees($monEntree){
                          $s = fgets($monEntree);
                          $s = fgets($monEntree);
                          $this->vitX = (int)$s;
                          $s = fgets($monEntree);
                          $this->vitY = (int)$s;
                          $s = fgets($monEntree);
                          $this->energie = (float)$s;
                          $s = fgets($monEntree);
             }
}

$unPredateur=new Predateur(10,10,100);
if (!($fp=fopen("leFichierPredateur.txt","w")))
```

```
{
            print("Ouverture impossible du fichier");
            exit;
}
stream_set_write_buffer($fp, 0);
$unPredateur->sauveDonnees($fp);
fclose($fp);
if (!($fi=fopen("leFichierPredateur.txt","r")))
{
            print("Ouverture impossible du fichier en lecture");
            exit;
}
$unPredateur->litLesDonnees($fi);
?>

</body>
</html>
```

Ce code PHP 5 est presque entièrement calqué sur le code Python. On y retrouve la même simplicité d’usage.

Sauvegarder les objets sans les dénaturer : la sérialisation

Bien que cette pratique nous assure que les informations à sauvegarder le seront effectivement, elle présente l’inconvénient de « sonner faux » dans une perspective plus OO. En effet, nul cas n’est fait de l’organisation en objet des informations à sauvegarder. Qu’ils soient contenus dans des objets ou non, ces entiers, doubles ou caractères à sauvegarder, le seront, d’une seule et même manière.

De plus, nous savons que les objets sont connectés entre eux, par un réseau relationnel qui peut, très vite, devenir important et complexe. Rien n’est prévu pour que la sauvegarde d’un des nœuds de ce réseau puisse déboucher sur la sauvegarde du réseau dans son entièreté. C’est au programmeur de penser la persistance, de manière à conserver également les liens qui relient les objets entre eux, lors de leur sauvegarde. L’utilisation de simples tubes à octet connectés au fichier ne le permet pas.

Si vous sauvegardez un attribut de type référent, vous vous limitez à sauvegarder une adresse, sans que l’objet adressé par cette adresse ne le soit également à un endroit correspondant sur le disque dur. En substance, il est nécessaire ici de recourir à un mode de sauvegarde plus puissant, qui permettrait, très facilement, de reproduire sur le disque dur une copie des objets et de leur structure d’interconnexion, c’est-à-dire de faire, à un instant donné, une copie miroir du tissu relationnel des objets qui se trouve dans la RAM. La lecture d’un des objets du réseau suffirait, pareillement, à la reproduction dans la RAM du réseau dans son entièreté. Ce mécanisme, absent du C++ standard (mais bien des librairies logicielles compensent aujourd’hui cette absence), est prévu en C#, Java, Python et PHP 5 et porte le nom de « sérialisation ».

Il suffit de découvrir la nouvelle version des méthodes `sauveDonnees()` et `litLesDonnees()` pour se convaincre de l’extrême simplicité de la mise en œuvre de la sérialisation.

En Java

```
import java.io.*;
public class Predateur extends Faune implements Serializable {
  private Proie[] lesProies;
```

```
  public void sauveDonnees(ObjectOutputStream oos) {
    try{
      oos.writeObject(this); /* on sauve l'objet */
    } catch (IOException e) { System.out.println(e); }
  }
  public void litLesDonnees(ObjectInputStream ois) {
    Predateur unPredateur;
    try{ /* on lit l'objet puis on le caste dans la classe adéquate */
      unPredateur = (Predateur)ois.readObject();
    } catch (Exception e) { System.out.println(e); }
  }
}
public class Jungle {
  …
    try {
      fis = new FileInputStream ("leFichierPredateur.ser");
      ois = new ObjectInputStream(fis); /* on installe un tube à lecture d'objets sur le fichier */
    } catch (IOException e) {e.getMessage();}
  for (int i=0; i<lesAnimaux.length; i++)
    lesLions[i].litLesDonnees(ois);
  try {
    ois.close(); /* on ferme le tube */
  } catch (Exception e) {}
  try {
    fos = new FileOutputStream("leFichierPredateur.ser");
    oos = new ObjectOutputStream(fos); /* on installe un tube à écriture d'objets sur le fichier */
  } catch (IOException e) {e.getMessage();}
  for(int i=0; i<lesLions.length; i++)
    lesLions[i].sauveDonnees(oos);
  try {
    oos.close(); /* on ferme le tube */
  } catch(Exception ex) { System.out.println(ex.getMessage()); }
}
```

Deux seules instructions s'occupent de tout : `oos.writeObject()` pour la sauvegarde des objets et `ois.readObject()` pour leur récupération. Malgré la simplicité déconcertante de l'approche, plusieurs points restent à éclaircir. D'abord, la lecture et l'écriture d'objets dans un fichier s'opèrent à l'aide des deux sous-classes de « stream » : `ObjectInputStream` et `ObjectOutputStream`. C'est sur les deux instances de ces deux classes que les messages de lecture et d'écriture d'objets sont envoyés. Pour l'écriture, l'objet est passé en argument. Dans sa déclaration initiale, la méthode `writeObject(Object unObjet)` peut recevoir comme argument n'importe quel objet de type Object, donc en final n'importe quel objet (une autre utilisation importante de la classe `Object` dont l'utilité a été discutée au chapitre 14).

De la même manière, la méthode `readObject()` est déclarée comme retournant n'importe quel type d'objet. Deux conséquences à cela. La première est qu'il faut recourir à un nécessaire « casting » (ici `Predateur`) pour utiliser cet objet comme il se doit. Cependant, à chaque utilisation du « casting », il est possible, et dommageable, que l'objet ne soit pas de la classe dans lequel on le « caste » ; il en résulte la génération d'une exception.

Il faut absolument prévenir cela, en intégrant la lecture de l'objet dans un `try-catch`. Vous pourriez, éventuellement, à partir de l'objet, et en utilisant les méthodes de réflexion comme `getClass()`, récupérer la classe de

l’objet (chaque objet possède un attribut caché qui est le nom de sa classe) et la tester avant d’effectuer le « casting ».

Toute classe qui souhaite que ses instances puissent être sérialisées doit implémenter l’interface `Serializable`. Ici, cette interface étiquette cette classe plutôt qu’elle ne lui indique un ensemble de services à implémenter, comme le font normalement les interfaces. La présence de `Serializable` indique juste que la classe se prête à cela et, s’il est nécessaire de l’expliciter, c’est que certaines classes peuvent ne pas pouvoir se sérialiser. Si vous créez une classe que vous pensez ne jamais devoir sérialiser, omettez d’implémenter cette interface.

Pensez par exemple à la classe `Thread`. Cela n’a pas de sens de sérialiser un objet `thread`, car il fait purement et simplement partie du contexte d’exécution du programme, et doit, à ce titre, être re-créé à chaque exécution. Il en va de même pour certaines classes de connexion aux périphériques et d’autres encore, comme la classe `Image`, dont les objets sont, là encore, par essence, temporaires.

La sérialisation est récursive, car en sauvegardant et en récupérant un objet, ici un prédateur, l’objet n’est pas le seul concerné, mais il emmène dans sa suite tous les objets qu’il réfère, et ces derniers, faisant de même, il se crée une chaîne d’objets à trimballer, comme des casseroles derrière une voiture de jeunes mariés. La sauvegarde se fait en « série », d’où son nom. Néanmoins, vous pourriez décider, par exemple, de ne pas sérialiser une classe, qui se trouve pourtant associée à une autre qui peut l’être.

De façon plus classique, une classe dont vous avez besoin dans la définition d’une autre (comme la classe `Thread` ou la classe `Image`) peut, par définition, n’être pas sérialisable. Java vous permet de contourner ce problème, en déclarant certains attributs « transient ». Les objets référés par ces attributs ne seront plus enchaînés aux autres, dans le long périple qui les mène jusqu’au disque dur. La version C# de ce mécanisme est très voisine, à quelques détails de syntaxe près (mais on s’y fait…).

En C#

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary; // Sérialisation en binaire
using System.Runtime.Serialization.Formatters.Soap; // Sérialisation en XML;

[Serializable] /* declare la classe sérialisable */

class Predateur {
  public void sauveLesDonnees(BinaryFormatter maSortie, Stream sw) {
    maSortie.Serialize(sw, this); /* on sauve l'objet */
  }
  public void litLesDonnees(BinaryFormatter monEntree, Stream sr) {
    Predateur unPredateur;
    /* on lit l'objet et on le caste dans la classe adequate */
    unPredateur = (Predateur)monEntree.Deserialize(sr);
  }
}
public class Jungle {
  public static void Main() {
    Stream sr = File.OpenRead("leFichierPredateur.ser"); /* on crée un tube à objet en lecture */
    BinaryFormatter monEntree= new BinaryFormatter();
    /* Il faut plutôt utiliser SoapFormatter pour la serialisation en XML */
```

```
    for (int i=0; i<lesAnimaux.length; i++)
      lesLions[i].litLesDonnees(monEntree, sr);
    sr.Close(); /* on ferme le tube */
    Stream sw = File.Create("leFichierPredateur.ser"); /* on crée un tube à objet en écriture */
    BinaryFormatter maSortie = new BinaryFormatter(); // ou SoapFormatter;
    for(int i=0; i<lesLions.length; i++)
      lesLions[i].sauveDonnees(maSortie, sw);
    sw.Close(); /* on ferme le tube */
  }
}
```

On retrouve dans la version C# plusieurs points communs avec Java. D'abord, la présence de `[Serializable]` (où l'on voit bien qu'il ne s'agit pas d'une vraie interface) devant la classe qui peut l'être, ainsi que la présence de `[NonSerialized]` devant un attribut, indiquant comme `transient` en Java que la sérialisation s'arrête à celui-ci. On ouvre un stream pour la lecture comme pour l'écriture, mais on recourt à l'utilisation de la classe `BinaryFormatter` pour le formatage des objets sauvés ou lus. La lecture et l'écriture se font au moyen des méthodes `Serialize` et `Deserialize` et, lors de la lecture, tout comme en Java, et pour les mêmes raisons, il est nécessaire de « caster » l'objet lu dans la classe d'accueil.

Notez qu'en C# et .Net en général, il est également possible de sérialiser très facilement des objets dans le format XML, soit en substituant simplement le `BinaryFomatter` par un `SoapFormatter` comme indiqué dans le code, soit par l'utilisation de la classe `XMLSerialize` et des méthodes `Serialize` et `Deserialize` recevant des « streams » en argument. Nous l'avons dit lors de la présentation du C#, Microsoft a intégré plus rapidement et naturellement le format XML au sein de ses applications et environnements de développement.

En Python

```
import pickle; #un des module qui permet la sérialisation
class Predateur(Thread):

      def sauveDonnees(self,maSortie):
              try:
                    pickle.dump(self.__uneEau,maSortie)
              except:
                    print "une erreur s'est produite"
      def litLesDonnees(self,monEntree):
              try:
                    x = pickle.load(monEntree)
              except:
                    print "une erreur s'est produite"

uneEau=Eau(1000)
unePlante=Plante(1000)
uneProie=Proie(uneEau,unePlante)
unPredateur=Predateur(uneEau,unePlante)
uneProie.start()
unPredateur.start()
unePlante.addObserver(unPredateur)
```

```
try:
      fr=open('leFichierPredateur.ser', 'w')
      unPredateur.sauveDonnees(fr)
      fr.close()
      fr=open('leFichierPredateur.ser', 'r')
      unPredateur.litLesDonnees(fr)
      fr.close()
except:
      print "une erreur s'est produite"
```

En Python, un certain nombre de modules traitent les opérations d’entrée-sortie qui permettent de sérialiser des objets. Nous avons choisi le module `pickle`, très simple à mettre en œuvre, comme le montre la présence des méthodes `dump` et `load`. Pas de problème de « casting » puisque Python ne nécessite pas le typage statique pour les objets traités.

PHP 5 donne à la sérialisation un sens un peu différent. Les opérations `serialize` et `unserialize` existent bel et bien dans ce langage, mais elle servent davantage à convertir dans un sens ou dans l’autre les objets en chaînes de caractères de manière à ce que ces mêmes objets soient (et là se trouve la connexion avec les autres langages) plus aisément stockables dans un fichier ou transmissibles via le réseau.

Contenu des fichiers de sérialisation : illisible

La simplicité du mécanisme de sérialisation dissimule une grande puissance et une formidable efficacité. Il suffit de comparer déjà les deux versions programmées de la persistance pour s’en rendre compte. Ce mécanisme rend parfaitement justice à la nature OO des données à sauver. De plus, il suffit de partir d’un nœud du réseau d’interaction entre objets pour qu’automatiquement, tous les autres soient pris en charge, en lecture comme en écriture.

Que manque-t-il donc à cette solution pour qu’elle s’impose comme la voie royale de la persistance des objets ? Une chose essentielle malheureusement, la lisibilité. Nous parlons évidemment ici de la version binaire de la sérialisation. Il est clair que la version XML discutée dans son implantation .Net permet de regagner en lisibilité ce que la version binaire avait perdu. Il ne suffit pas de sauvegarder les objets sur le disque dur, il faut encore qu’ils soient accessibles et lisibles sans devoir, obligatoirement, relancer l’exécution d’un programme Java ou C#. Il faudrait, comme vous pourriez le faire avec le fichier texte de la première version de la persistance, simplement ouvrir le fichier et y lire les objets et leur état. Or, observez ci-après le fichier `leFichierPredateur.ser`, quand il est ouvert avec le bloc-note de Windows.


```
« ¬í _sr  Predateur}'√% Ï¬µ_ _L _fost _Ljava/io/FileOutputStream;[  lesProiest _[LProie;xr _
Fauneææõ_‘V_√_ _D energieZ _repereI _vitXI _vitY[
lesRessources »
```


Vous y comprenez quelque chose ? Au pire, on devine certaines choses. Seul un programme Java peut vraiment lire les objets, vous les traduire et, éventuellement, afficher leur contenu. Il en va exactement de même pour C# ou Python (dans leur version binaire uniquement, la version SOAP permise par C# nécessite l’utilisation d’un parseur XML dédié pour accéder à l’information désirée). Il s’agit donc bien d’un mécanisme de sauvegarde des objets, mais dont Java, C# ou Python se réservent la complète exclusivité. Or, si on peut facilement admettre qu’il n’y a aucune raison de se pencher sur les fichiers sauvegardés lors de la dernière partie de votre jeu favori, sans avoir, d’abord, relancer le jeu, c’est plus difficile à admettre pour des comptes en banque ou autres informations, auxquels on devrait pouvoir accéder entre deux sessions d’exécution de Java, de C# ou de Python.

En outre, il existe aujourd'hui un mode de dépôt des données sur disque dur, mode très lisible, optimisé, très efficace, et surtout qui s'est extraordinairement répandu : les bases de données. Pour couronner le tout, ces bases de données ressemblent comme deux gouttes d'eau au modèle OO. Hélas, des gouttes qui restent très troubles, car si l'équivalence table/classe et enregistrement/objet est plus que tentante, elle est loin d'être vérifiée.

Les bases de données relationnelles

Les bases de données relationnelles sont, aujourd'hui, le moyen le plus répandu pour stocker l'information de manière permanente. Au grand dam de ceux qui aimeraient pousser la vision « objet » jusqu'au fin fond du disque dur, par exemple, par un mécanisme de sérialisation plus transparent, et acceptant une lecture des objets sans lancer un programme, il y a peu de raison que cela change dans les années à venir. C'est pour cela que les programmes OO doivent s'interfacer avec ce mode de stockage, pour y entreposer leurs objets, entre deux phases d'utilisation.

Les raisons du quasi-monopole de ce mode de stockage sont nombreuses. Une première en est qu'à l'instar de l'approche OO, les données sont concentrées dans des structures de type « objet ». Les informations à stocker le sont en tant que valeurs d'attribut, regroupées dans une structure appelée « table », très voisine de nos classes. Chaque instance de cette table, obtenue en fixant les valeurs d'attribut pour un cas donné, et qui donnerait naissance à un objet dans le cas de la classe, donne ici naissance à un enregistrement. Nous verrons les différences essentielles que présentent tables et classes dans la suite du chapitre.

Une deuxième raison est l'existence d'un mode d'organisation de ces tables, mode dit relationnel, et dont la qualité première est d'éviter d'avoir à reproduire la même information plusieurs fois. On imagine aisément tous les problèmes liés à la duplication des informations : mises à jour plus pénibles et erreurs d'encodage plus probables. Chaque table se doit de coder un concept donné, et il est nécessaire de séparer l'encodage des informations dans des tables distinctes, mais conservant entre elles des relations qui permettent, à partir de l'une d'entre elles, de retrouver les informations de l'autre.

Nous verrons que l'existence de ces relations entre les tables, conceptuellement proche des liens d'association entre les classes, intensifie davantage encore la ressemblance avec l'approche OO. C'est pourtant dans le mécanisme qui concrétise ces relations que réside toute la différence entre les bases de données relationnelles et la manière dont les objets existent et se réfèrent entre eux dans la mémoire de l'ordinateur.

Une troisième raison est la quantité importante de solutions techniques disponibles aujourd'hui, pour gérer de façon automatique des problèmes aussi critiques que les « backups automatisés », les accès concurrentiels (quand plusieurs accès doivent s'opérer simultanément), les accès sécurisés et fiabilisés, les accès de type transactionnel (quand une étape défaillante dans une succession d'opérations rend caduques toutes les opérations effectuées jusque-là), le stockage réparti sur plusieurs ordinateurs, etc.

SQL

Une dernière raison est l'existence d'un langage d'interrogation de ces bases de données, langage standardisé nommé SQL (Structured Query Langage), et que nous utiliserons dans la suite. Toute application informatique qui doit s'interfacer avec une base de données relationnelle le fera en « parlant » ce langage. C'est ce standard, additionné à toutes les solutions techniques proposées par des environnements logiciels, bien installés aujourd'hui, comme Oracle, Informix, Sybase ou Access, additionné, enfin, à toutes les couches logicielles de

visualisation et d'analyse de données s'exécutant à partir de ces bases, qui fait les bases de données relationnelles deviennent incontournables lors de l'étude des solutions de persistance des objets.

SQL

SQL est un langage d'interaction avec les bases de données qui sont utilisables, généralement, dans des programmes informatiques qui interagissent avec ces bases. Ces instructions permettent la création des tables et des relations, la consultation, l'insertion, l'effacement et la modification des enregistrements, la définition des permissions au niveau des utilisateurs.

Select pour lire les enregistrements :

```
SELECT [ALL] | [DISTINCT]
<liste des noms de colonnes> | *
FROM <Liste des tables>
[WHERE <condition logique>]
```

Insert pour insérer des nouveaux enregistrements :

```
INSERT INTO Nom_de_la_table
(colonne1,colonne2,colonne3,...)
VALUES (Valeur1,Valeur2,Valeur3,...)
```

Update pour mettre à jour ces enregistrements :

```
UPDATE Nom_de_la_table
SET Colonne = Valeur
[WHERE qualification]
```

Delete pour effacer ces enregistrements :

```
DELETE FROM Nom_de_la_table
WHERE qualification
```

Une table, une classe

Supposons, dans un premier temps, que nous souhaitions sauvegarder les quatre informations de notre prédateur, introduites dans les chapitres précédents, dans une table dénommée `Predateur`, dans la base de données Microsoft Access.

Figure 19-1

En Access, la table prédateur.

idPredateur	vitx	vity	energie
0	15	0	0,005
1	17	0	0,006
2	6	0	0,02
3	0	0	0,0001
4	2	0	0,063

Parmi les quatre attributs indiqués dans la table, pour les cinq prédateurs qui y sont enregistrés, un n'apparaîtrait pas dans l'approche OO, il s'agit de la « clé primaire », nommée ici `idPredateur`. On conçoit bien l'existence d'un tel attribut dans une base de données. C'est par lui que l'on accède à tous les enregistrements désirés.

Le système de gestion de la base de données sait, à partir d'une valeur donnée de cet attribut, retrouver l'enregistrement unique correspondant.

> **Clé primaire**
>
> La clé primaire d'une table est l'attribut qui permet d'accéder, de manière unique, et sans aucune équivoque à un des enregistrements de la table. La valeur de cet attribut doit être distincte pour chaque enregistrement, une valeur entière étant ce qu'il y a de plus courant.

Pourquoi, dès lors, cet attribut disparaît de la classe `Predateur` ? Car la classe `Predateur` n'a nullement besoin d'intégrer dans sa définition un mode d'accès à ses instances. Chaque instance est retrouvée par l'utilisation d'une variable « référent », dont la valeur est l'adresse physique de cette instance. N'oublions pas que l'OO est d'abord et avant tout un mode de programmation, manipulant un ensemble de variables permettant de référer des informations stockées dans la mémoire vive, alors que les bases de données relationnelles sont, avant tout, un simple mais puissant mode de stockage de l'information, fiable et économique, dans lequel ce mécanisme de référents n'a plus lieu d'être.

> **Référent versus clé**
>
> Les bases de données relationnelles fonctionnent et s'utilisent indépendemment de tout mode de stockage, là où les objets ne s'utilisent et ne fonctionnent que sur un mode stocké. Un enregistrement est indicé par un nombre unique plutôt qu'un emplacement unique. C'est là toute la différence, et ce n'est pas rien.

Dans les deux cas, l'objet `predateur` et l'enregistrement `predateur`, il est nécessaire de pouvoir retrouver l'information sans équivoque aucune. Tant la clé primaire que le référent permettent cela. Cependant, la clé primaire n'a pas obligation de correspondre à une adresse physique, un système de gestion de la base de données s'occupant de mémoriser la correspondance entre la valeur de cette clé et l'emplacement de l'enregistrement.

Chaque prédateur, dans une base de données relationnelle, doit connaître son identifiant unique, comme vous pourriez connaître le numéro qui vous identifie dans votre passeport ou carte d'identité. Les objets prédateurs ne le doivent pas, car leurs utilisateurs, connaissant leur adresse, savent parfaitement les retrouver. L'objet existe à ce point pour autrui que l'absence de référents sur un de ces objets revient à signer son arrêt de mort.

Comment Java et C# s'interfacent aux bases de données

Voyons maintenant, en Java et en C# comment les deux méthodes `sauveDonnees() et litLesDonnees()` du prédateur sont redéfinies, dans le cas où toute l'information concernant ce prédateur est stockée dans une table Access. C++ n'a pas intégré dans sa syntaxe de base l'accès aux bases de données. Tout ce que nous ferons dans la suite en Java et en C# pourrait bien évidemment être réalisé de la même manière dans C++, mais en se reposant entièrement sur des fonctionnalités additionnelles, proposées par le système d'exploitation et le « pilote » de la base de données. Nous passerons donc sous silence l'interaction entre C++ et les bases de données. C++ n'a pas ce même souci d'universalité partagé par Java et C# (ce dernier, pour autant que l'univers s'arrête aux systèmes d'exploitation Windows).

Quant à Python, sa bibliothèque standard ne fournit pas à ce jour d'interface universelle à toutes les bases de données relationnelles. Cependant, de nombreux modules tiers permettent aux programmes de se connecter à différentes bases de données (comme DB2, MySQL, Informix, Microsoft SQL Server, Adabas, Sybase…).

Nous laisserons donc pour le moment la version Python de côté, conscients du fait que de multiples solutions d'interfaçage aux bases de données relationnelles existent, qui permettraient de reproduire sans difficulté aucune les versions Java et C#.

PHP 5, pour sa part, a parfaitement intégré l'interaction avec les bases de données (et ceci depuis ses toutes premières versions), puisqu'il s'agit là d'une des fonctionnalités essentielles des scripts web qui s'exécutent côté serveur. Nous le passerons cependant sous silence car il s'assimile pour l'essentiel aux versions Java et C# et l'interaction entre les bases de données et le PHP (le couple PHP/MySQL est bien connu pour la conception de sites web dynamiques) a déjà fait l'objet de très nombreux ouvrages.

En Java

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import java.sql.*;
public class Predateur extends Faune {
  public void litLesDonnees(Connection connexion, int i) {
    try {
      Statement ordre    = connexion.createStatement(); /* établit la connexion */
      String requete     = "SELECT * FROM Predateur WHERE IdPredateur =" + i; /* la requête SQL */
      ResultSet resultats= ordre.executeQuery(requete); /* envoie la requête sur la connexion */
      resultats.next(); /* itère le résultat */
      setVitX(resultats.getInt("vitx"));
      setVitY(resultats.getInt("vity"));
      setEnergie(resultats.getDouble("energie"));
      ordre.close();
    } catch (Exception e) { System.out.println(e.getMessage()); }
  }
  public void sauveDonnees(Connection connexion, int i) {
    try {
      Statement ordre = connexion.createStatement();
      /* la requête SQL */
      String requete = "UPDATE Predateur SET " + "vitx='" + getVitX() + "', vity='" + getVitY()
                     + "', energie='" + getEnergie() + "' WHERE IdPredateur =" + i;
      System.out.println("j'execute la requete");
      int resultat = ordre.executeUpdate(requete); /* envoie la requête sur la connexion */
      if (resultat == 1)
        System.out.println("la base de donnees est mise a jour");
      else
        System.out.println("c'est rate");
      ordre.close();
    } catch (Exception e) { System.out.println(e.getMessage()); }
  }
}
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
```

```
import java.sql.*;
public class Jungle {
  private Connection connexion;

  public Jungle(int largeur, int hauteur) {
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); /* va chercher le pilote adéquat */
      /* établit la connexion sur la base */
      connexion = DriverManager.getConnection("jdbc:odbc:Ecosysteme");
    } catch (Exception se) {System.out.println("Connexion Impossible" + se.getMessage());}
    for (int i=0; i<lesAnimaux.length; i++)
      lesLions[i].litLesDonnees(connexion,i);
    for(int i=0; i<lesLions.length; i++)
      lesLions[i].sauveDonnees(connexion, i);
  }
}
```

Commençons cette fois par la classe `Jungle`. Elle doit créer la connexion vers la base de données. Elle le fait en créant un objet de type `Connection`, qui sera passé comme argument des méthodes `sauveDonnees()` et `litLesDonnees()`. L'objet `Connexion` nécessite, d'abord, l'activation d'un pilote vers une base de données de type « Access ». L'instruction `Class.forName(nomDeClasse)` permet de charger en mémoire tout ce qui concerne la classe indiquée entre parenthèses, toutes les méthodes de cette classe et les attributs statiques, bref, tout ce qui est exploitable sans devoir créer la moindre instance de cette classe.

Ici, cette instruction met à disposition ce qui nous permettra de « dialoguer » avec notre base de données Access. ODBC est un pont vers les bases de données Microsoft qui permet de recevoir et d'exécuter des requêtes SQL sur ces bases de données.

Ayant encore et toujours vocation d'universalité (on se répète…), Java doit, de son côté, créer une sorte de pont équivalent, appelé « JDBC », mais qui permette, cette fois, à partir d'un code, de pouvoir exécuter n'importe quelle requête SQL sur n'importe quelle base de données relationnelle. JDBC s'est très largement inspiré d'ODBC. Lorsqu'il s'agit d'une base de données Microsoft, comme c'est le cas d'Access, il reste à connecter les deux ponts (bien que le protocole JDBC soit extrêmement proche d'ODBC), et c'est le rôle du `sun.jdbc.odbc.JdbcOdbcDriver`.

Java possède également dans ses libraires de quoi établir des ponts JDBC vers la majorité des bases de données relationnelles qui, de leur côté, ont développé les pilotes adéquats. Tout cela, en fait, ne sert qu'à véhiculer des requêtes SQL, du code Java vers les bases de données, et d'en recevoir les résultats dans une forme lisible et exploitable (souvent, un vecteur d'objets sur lequel tout est prévu pour la lecture des composants).

La connexion avec la base de données spécifique, que nous avons appelée `Ecosysteme`, s'établit au moyen de l'instruction suivante : `connexion = DriverManager.getConnection("jdbc:odbc:Ecosysteme")`, opération encadrée par un **`try-catch`**, on devine aisément pourquoi. C'est sur cet objet de type `Connection` que nous enverrons les requêtes, et c'est donc lui que nous passons en argument des méthodes `litLesDonnees()` et `sauveDonnees()` du prédateur.

Commençons par la méthode `litLesDonnees()`, qui récupère des informations en provenance de la base de données. Il faut d'abord créer un ordre, sur lequel nous exécuterons notre requête. Cet ordre est un objet de la classe `Statement`, obtenu à partir de l'objet `Connexion`. Cela permet de relier l'ordre à la base de données qui nous intéresse. Toute requête SQL est une chaîne de caractères, que nous associerons ici à « l'ordre » par la

méthode executeQuery(). Il s'agit, dans un premier temps, d'une simple lecture qui retourne un certain résultat. Cette requête sera notre première requête SQL, un SELECT.

Il n'est pas question ici de détailler SQL. La place nous manque (excuse classique) et il suffit de savoir qu'à partir des instructions issues de ce langage, il est possible de créer des tables, de rajouter de nouveaux enregistrements (INSERT), de modifier certains enregistrements (UPDATE), de lire les enregistrements (SELECT), etc. Ce langage est puissant, standard, et vite assimilé, car logiquement conçu. Par exemple, la requête SELECT, que nous envoyons vers la base de données Ecosysteme, lit toutes les valeurs d'attribut du énième prédateur (la forme de cette requête doit vous paraître assez compréhensible).

Le résultat est envoyé sur un nouvel objet de type ResultSet, une collection d'éléments qui permet, très simplement, de passer en revue ces différents éléments au moyen de la méthode next(). La dernière opération consiste à extraire les valeurs d'attributs qui nous intéressent par la méthode getInt(), lorsque nous savons qu'il s'agit de lire un entier, ou getDouble() lorsqu'il s'agit de lire un réel. Tout cela, bien sûr, en se préoccupant de toutes les exceptions que ces manœuvres pourraient lever (base indisponible, accès non autorisé, getInt() appliquée sur autre chose qu'un entier…).

La méthode sauveDonnees(), quant à elle, enverra sur l'objet Ordre une requête de type UPDATE, puisqu'il s'agit de mettre à jour les enregistrements de la table Predateur avec les nouvelles valeurs des attributs. La requête UPDATE … SET …. WHERE permet de modifier uniquement certains attributs (indiqués dans le SET) d'une table (indiquée dans l'UPDATE) pour les enregistrements vérifiant la condition WHERE. Elle est envoyée sur l'objet Ordre par l'entremise de la méthode executeUpdate(), différente de executeQuery(), en ceci qu'elle n'attend rien en retour.

Enfin, que ce soit lors de la lecture ou de l'écriture, il sera nécessaire, une fois l'une ou l'autre effectuée, de fermer les connexions établies avec la base de données, au risque que les transmissions ne s'effectuent pas correctement.

En C#

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Data;
using System.Data.OleDb;
class Predateur : Faune {
  public void litLesDonnees(OleDbConnection connexion, int i) {
    try {
      string requete = "Select * FROM Predateur WHERE IdPredateur =" + i;
      OleDbCommand ordre = new OleDbCommand(requete, connexion);
      Console.WriteLine("j'execute la requete");
      OleDbDataReader resultats = null;
      resultats = ordre.ExecuteReader(); /* exécute la requête SQL */
      string s = null;
      resultats.Read();
      s = "" + resultats["vitx"];
      setVitX(Int32.Parse(s));
      s = "" + resultats["vity"];
      setVitY(Int32.Parse(s));
      s = "" + resultats["energie"];
```

```
      setEnergie(Double.Parse(s));
      Console.WriteLine(vitX);
      resultats.Close();  /* ferme le tube */
    } catch (Exception e) {Console.WriteLine("c'est rate " + e);}
  }
  public void sauveDonnees(OleDbConnection connexion, int i) {
    try {
      String requete = "UPDATE Predateur SET " + "vitx='" + vitX + "', vity='" + vitY + "', energie='
      ➥" + energie + "' WHERE IdPredateur =" + i;
      Console.WriteLine("j'execute la requete");
      OleDbCommand ordre = new OleDbCommand(requete, connexion);
      ordre.ExecuteNonQuery(); /* exécute la requête SQL */
      ordre.Close();
    } catch (Exception e) {Console.WriteLine("C'est rate " + e);}
  }
}
public class Jungle {
  public static void Main() {
    /* établit la connexion avec la base de données */
    string strConnection = "Provider=Microsoft.Jet.OleDb.4.0;";
    strConnection +=     @"DataSource=C:\Persist\Persist\Ecosysteme.mdb"; /* le "@" permet de
      ➥conserver les caractères spéciaux */
    OleDbConnection connexion = new OleDbConnection(strConnection);
    try {
      connexion.Open();
      Console.WriteLine("Connexion reussie");
      for (int i=0; i<lesAnimaux.length; i++)
        lesLions[i].litLesDonnees(connexion,i);
      connexion.Close();
      Console.WriteLine("Connexion fermee");
    } catch (Exception e) {Console.WriteLine("C'est rate" + e);}
    try {
      connexion.Open();
      Console.WriteLine("Connexion reussie");
      for(int i=0; i<lesLions.length; i++)
        lesLions[i].sauveDonnees(connexion, i);
      connexion.Close();
      Console.WriteLine("Connexion fermee");
    } catch (Exception e) {Console.WriteLine("C'est rate" + e);}
  }
}
```

Dans la version C#, le pont vers la base de données « Ecosystème » s'établit par un jeu d'instructions qui met en œuvre la classes `OleDb`, propre à l'architecture .Net. Une fois la connexion établie, l'esprit (mais pas la lettre malheureusement) est très proche de Java. En effet, comme en Java, cet objet `Connexion` est passé en argument des méthodes `litLesDonnees()` et `sauveDonnees()`.

Comme en Java également, un ordre est créé sur lequel s'exécutera la méthode `executeReader()` pour la lecture *via* un `SELECT`, et `executeNonQuery()` pour l'écriture sur la base *via* un `UPDATE` (en l'absence d'un renvoi de résultat). Dans le cas du `SELECT`, toujours comme en Java, le résultat est passé dans une classe `collection`,

ici de type `OleDbDataReader`, sur laquelle une lecture itérée est possible par la méthode `Read()`. Les valeurs d’attributs sont récupérées en tant que « string », et il est nécessaire de les transformer en entier ou double.

Depuis la version .Net 2, la nouvelle version du code C# s’écrira plutôt comme suit :

```
using System;
using System.Data;
using System.Data.Common;

public class TestDB {
   public static void Main() {
      DataTable dt = DbProviderFactories.GetFactoryClasses();
      DbProviderFactory dbpf = DbProviderFactories.GetFactory
              ("System.Data.OleDb");
      DbConnection connexion = dbpf.CreateConnection();
      connexion.ConnectionString =  "Provider=Microsoft.Jet.OleDb.4.0;";
      connexion.ConnectionString+=   @"DataSource=C:\Persist\Persist\Ecosysteme.mdb";;
      String requete = "SELECT * FROM Predateur WHERE IdPredateur = " + i;
      DbCommand ordre = connexion.CreateCommand();
      ordre.CommandText = requete;
      DbDataReader resultat = null;
      connexion.Open();
      resultat = ordre.ExecuteReader();
      resultat.Read();
   }
}
```

Toutes les classes débutant par `OleDb` se doivent maintenant de débuter par `Db`. En substance, .Net 2 a enrichi son mode d’accès aux bases de données relationnelles d’une couche logicielle additionelle qui lui permet de généraliser un même code à plusieurs type de bases de données. Ce code fonctionnera de manière identique quel que soit le pilote de connexion à la base de données, qu’il soit de type `OleDb` ou pas.

C’est l’instruction `DbProviderFactory dbpf = DbProviderFactories.GetFactory("System.Data.OleDb")` qui va particulariser ce code générique au cas du driver `OleDb`.

Relations entre tables et associations entre classes

Tant que nous nous limitons à une table et à une classe, la sauvegarde et la récupération des objets à partir d’une base de données relationnelle semblent une opération bien bénigne, qui se règle facilement à coup de quelques requêtes SQL bien maîtrisées. Malheureusement, il n’est pas possible d’avoir affaire à un programme OO à une classe : un programme OO, c’est toujours un réseau de classes qui, de manière coresponsable, s’occupent de mener à bien l’application.

Qu’à cela ne tienne direz-vous, les tables peuvent être tout autant en relation que les classes le sont, le passage de l’OO vers le relationnel ne se trouve-t-il pas facilité par cette ressemblance structurelle ? Quelques requêtes SQL ne suffisent-elles pas à mener à bien le va-et-vient des objets entre le programme et la base de données, quelles que soient les liaisons dangereuses qui existent entre ces objets ?

Relation 1-n

Pour répondre à cette question, nous envisagerons quelques relations, une par une, pouvant exister entre les classes de notre écosystème, et nous nous pencherons sur leur traduction dans le mode relationnel.

Tout d'abord, supposons une nouvelle classe, `Taniere`, habitat de notre prédateur, et admettons, de surcroît, que chacun des prédateurs peut se loger dans plusieurs de ses tanières.

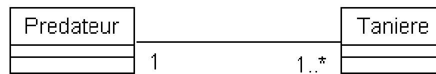


Figure 19-2
Relation 1-n entre la table prédateur et la table tanière.

Dans l'approche OO, le prédateur interagit avec ses tanières en possédant un vecteur de référents vers celles-ci, la valeur de chaque référent étant l'adresse d'un des objets tanières, avec lequel le prédateur peut interagir. Chacun des prédateurs a donc une connaissance directe de ses tanières, il possède l'adresse de chacune d'entre elles. Dans l'approche base de données relationnelle, il y a lieu, tout comme en OO, de tenir séparé ces deux tables, puisqu'elles concernent, en effet, deux réalités très distinctes.

Cependant, il n'est plus question ici d'un adressage physique explicite, mais simplement d'une manière, pour chaque enregistrement de la première table, de découvrir avec quels enregistrements de la seconde table, il entre en relation. Cela se fait, tout simplement, en rajoutant à la seconde table une clé dite « étrangère », dont les valeurs doivent être partagées avec celles prises par la clé primaire de la première table.

La relation s'établit alors, comme indiqué dans la figure ci-après, entre la clé primaire de la première table et la clé étrangère de la seconde. La tanière saura de quel prédateur elle est la tanière, par l'entremise de cette clé étrangère, dont la valeur correspondra à une des valeurs de la clé primaire des prédateurs. Il existe une façon de forcer ce partage des valeurs entre les deux clés, en imposant l'intégrité référentielle dans la relation entre ces deux tables. Les tanières ne peuvent être alors les tanières que des seuls prédateurs existants.

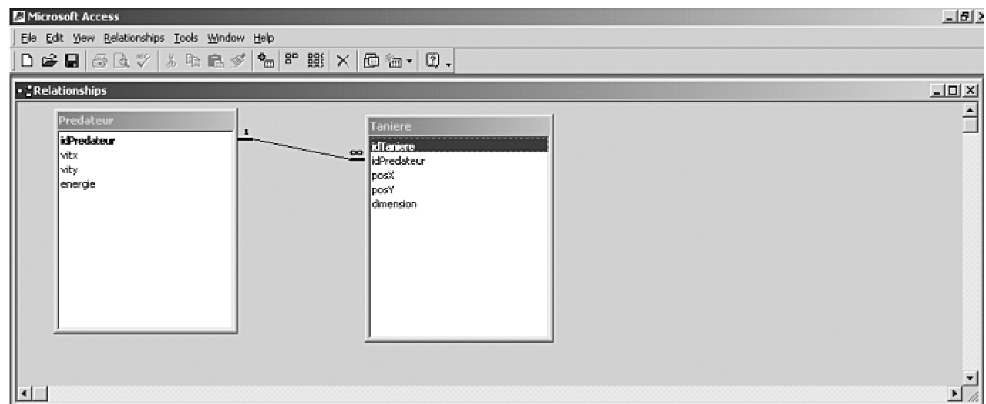


Figure 19-3
Concrétisation par le jeu des clés primaires et étrangères de la relation 1-n.

Ici, la liaison entre le prédateur et la tanière se fait par une recherche de valeurs communes entre les clés primaires et étrangères, et nullement par un adressage explicite, comme dans l'approche OO. À un même type « conceptuel » de relation correspondent deux mécanismes de mise en relation complètement différents. Le relationnel et l'OO, c'est un peu l'histoire de Gini et de l'alcool. Cette différence fonctionnelle est à la base du « clash culturel » entre le monde relationnel et le monde OO.

Ce clash n'est pas une aubaine pour la communauté informatique qui, aujourd'hui, a adopté, dans une large partie, la pratique OO comme pratique de programmation, et la pratique relationnelle comme pratique de

sauvegarde de données. Il suffit de programmer la moindre interaction entre un code OO et une base de données relationnelles pour se rendre compte des nombreux problèmes pratiques et des incroyables migraines que provoque l'intégration des requêtes SQL à même le code. Nous aurons l'occasion de le constater très vite dans le problème de réservation de places de spectacle qui suit. Ce clash trouve son origine dans une justification différente de l'existence de ces relations entre tables ou classes.

Pour ce qui est des classes, nous nous trouvons en pleine logique de programmation : l'ordinateur exécute des instructions, et deux classes doivent se parler, dès lors que la première déclenche sur l'autre une séquence d'instructions. Pour ce faire, elle n'a d'autre moyen, plus direct, que de connaître l'adresse des données que ces instructions doivent manipuler.

Pour ce qui est des tables, nous sommes en pleine logique de stockage de données, l'important étant la facilité d'utilisation, d'écriture et de lecture de ces données. Les relations sont parfaitement justifiées lors de l'écriture ou de la mise à jour des données, car elles permettent d'éviter des redondances malvenues, souvent sources d'erreur lors de l'écriture ou de cette mise à jour. Ces relations existent, alors, sans aucun besoin d'emplacement physique. Elles existent à titre « conceptuel », que les tables soient stockées sur un disque dur ou dans un frigidaire.

En conséquence de quoi, on se rend compte que, dès qu'un diagramme de classe intègre une relation de type 1-n entre deux classes (et c'est plutôt fréquent), il est indispensable de penser une traduction dans le mode relationnel. Cette traduction, idéalement, passe par l'addition, en plus des attributs issus des deux classes, d'une clé primaire dans une des tables et d'une clé étrangère dans l'autre, ainsi que par une assignation de valeur pour ces clés, qui traduisent les relations existant entre les objets.

En substance, la figure 19-3 devrait être une traduction automatisée de la figure 19-2. La version relationnelle sera souvent plus contrainte que la version OO, dans la mesure où, dans la première, aucune tanière ne peut exister sans faire référence à un prédateur (à cause de l'intégrité référentielle), alors qu'il pourrait y avoir dans la seconde des tanières inhabitées (la flèche d'association dans le diagramme UML serait alors dirigée). Les instructions SQL se complexifieront également quelque peu, car elles devront prendre en considération l'existence de ces relations. Par exemple, si l'on désire accéder à la dimension de la seconde tanière du premier prédateur, la requête SQL le permettant sera la suivante (INNER JOIN indiquant la jointure entre les deux tables) :

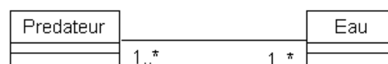
```
SELECT Taniere.dimension FROM Predateur INNER JOIN Taniere ON
Predateur.idPredateur = Taniere.IdPredateur WHERE Predateur.idPredateur = 1 AND idTaniere = 2
```

Alors que son équivalent OO se limiterait à quelque chose comme : `Predateur[0].getMesTannieres()[1].getDimension()`. En fait, tout cela pour dire qu'il faut quelque peu « tordre le cou » à l'approche OO pour coller à la vision relationnelle et, encore, nous ne sommes pas au bout de nos peines ni de nos triturations. Dans le diagramme de classe, l'association qui existe entre le prédateur et, disons, les points d'eau (nous ne considérons pas d'héritage pour l'instant) est de type n-n. En effet, plusieurs prédateurs peuvent s'abreuver à un même point d'eau et un prédateur quelconque peut s'abreuver à plusieurs points d'eau.

Relation n-n

Figure 19-4

Relation n-n entre les prédateurs et les points d'eau.



Est-il possible de reproduire cela, de la manière la plus simple, par une relation équivalente, `n-n`, entre les deux tables. Ne vous y risquez pas ! C'est parfaitement prohibé étant donné le mécanisme de jointure qui sous-tend les relations `1-n`, et cela contribuerait à dégrader considérablement nos relations, entre vous et nous, cette fois… La table du côté du n comprend une clé étrangère à associer à la clé primaire de la table du côté du 1. Une seule clé étrangère est

nécessaire, car chaque enregistrement de cette table n'est associé qu'à un seul enregistrement de l'autre table, quitte à ce que cela soit plusieurs fois le même (pour que l'on ait bien « 1 » d'un côté et « n » de l'autre).

Or, si nous voulions une relation de type n-n, il faudrait permettre plusieurs clés étrangères dans la table du côté du n, pour que chaque « eau » puisse référer plusieurs prédateurs. C'est parfaitement impensable, car nous ne saurions combien en mettre et comment se ferait la jointure entre la clé primaire et ces multiples clés étrangères.

En conclusion, les relations n-n ne sont pas admises dans les bases de données relationnelles, alors qu'il peut en exister, sans le moindre problème, en OO. On conçoit bien qu'un premier objet puisse pointer vers de multiples objets, et que ces derniers puissent faire de même. Cette relation `n-n` nous permet de comprendre davantage encore les deux mécanismes très différents mis en œuvre pour relier les tables et les classes entre elles : lien conceptuel d'un côté, lien physique de l'autre. Le seul moyen de s'en sortir dans l'univers relationnel est de procéder, comme montré dans la figure ci-après, à l'ajout d'une table intermédiaire, une table qui associe les points d'eau aux prédateurs.

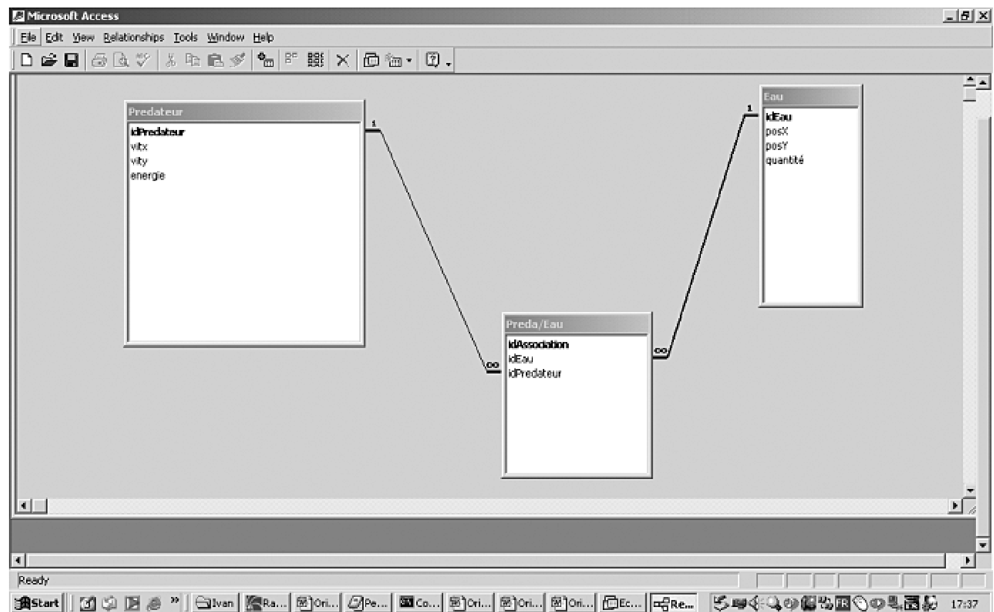


Figure 19-5
Décomposition d'une relation n-n en deux relations 1-n.

Cette table additionnelle, sans équivalent dans la version OO, complexifie encore la traduction du monde objet vers le relationnel, inévitable pour pouvoir sauvegarder les objets et leurs relations dans une base de données. Il faudra, *via* les requêtes SQL, créer, accéder et modifier cette table d'association, alors qu'elle ne paraît être la contrepartie sémantique d'aucune des classes en présence.

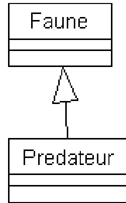
Dernier problème : l'héritage

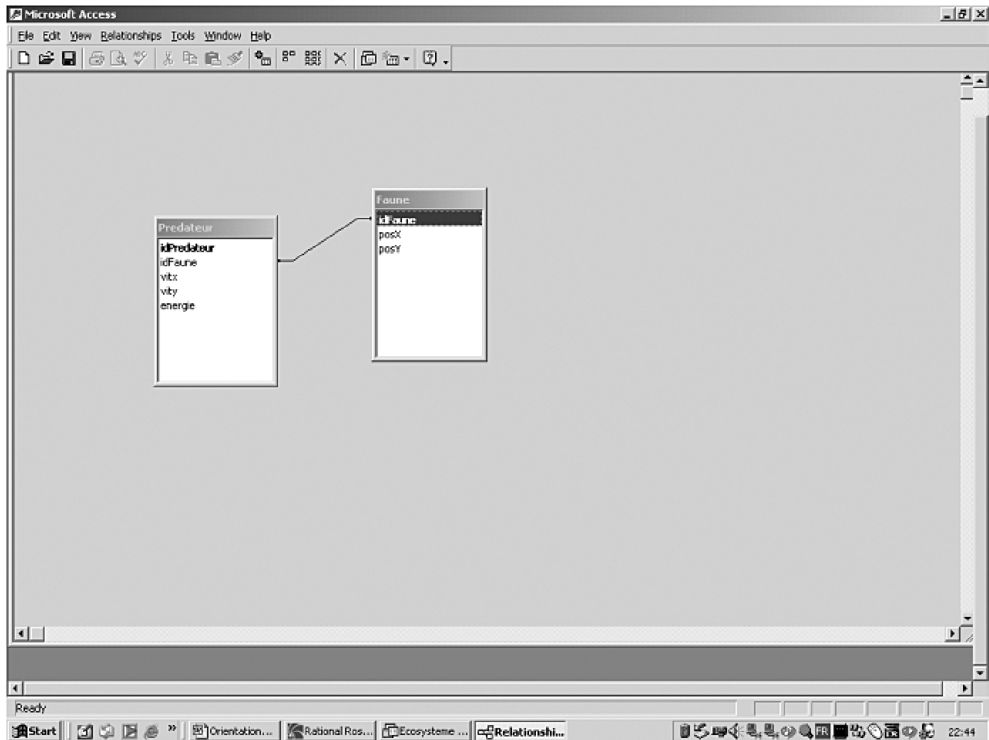
Cette relation d'héritage entre les classes n'existe pas dans les bases de données relationnelles. Une table ne peut hériter d'une autre. Pour autant, et de manière assez paradoxale, un même souci d'économie et de non-redondance prévaut à l'existence de l'héritage entre les classes et aux relations entre les tables. En effet, de nombreux attributs de notre prédateur sont hérités de la classe `Faune`, dans un souci d'économie. Motivée par

une semblable préoccupation, la traduction de l'héritage en une relation 1-1 entre les deux tables peut s'effectuer très simplement comme indiqué ci-après.

Figure 19-6

Traduction d'une relation d'héritage entre classes en une simple relation entre tables.





Chaque enregistrement prédateur est en relation `1-1` avec un enregistrement faune, qui reprend toutes les valeurs d'attributs dont le prédateur hérite dans la version OO. Certains adeptes du monde relationnel n'admettent pas les relations `1-1` car ils argumentent, à raison, qu'une seule table joignant les attributs des deux précédentes ferait aussi bien l'affaire. Du point de vue des performances, cette attitude se justifie, mais elle est plus discutable du point de vue « sémantique ». Cependant, nous voyons bien qu'une traduction OO/ relationnel est toujours possible. Certains vendeurs de bases de données relationnelles, acteurs du monde de l'OO, proposent des outils logiciels qui automatisent cette traduction, par exemple, en automatisant la génération du schéma relationnel à partir du diagramme de classe, ainsi que les requêtes SQL qui permettent de lire et d'écrire l'état des objets.

Bien sûr, le résultat n'est pas toujours des plus digestes, et le programmeur OO aura comme une impression dérangeante de dénaturation de son programme, par rapport à la simple sérialisation, qui permet de sauvegarder et de lire les objets en un coup de cuillère à POO. De plus, cette traduction ne fait aujourd'hui l'objet d'aucune vraie standardisation du côté de l'OMG.

Réservation de places de spectacles

Parmi les exercices situés à la fin du chapitre 10 consacré à l'UML, nous trouvons l'énoncé suivant. Vous devez réaliser un programme s'occupant de la gestion des réservations d'une salle de spectacle. Votre programme vend des réservations pour une représentation (un certain jour, à une certaine heure) d'un spectacle (caractérisé au minimum par son auteur, son titre et son type). Un client, identifié par ses nom, prénom, adresse et numéro de téléphone, peut effectuer autant de réservations qu'il le souhaite. Selon que le client est un abonné ou non (auquel cas le système conserve l'année de son inscription), il bénéficie d'une priorité sur les réservations (il peut réserver des places une ou deux semaines avant les non-abonnés) et de facilités de paiement (il peut payer l'ensemble de ses réservations à la fin de l'année s'il le désire). Une fois la réservation enregistrée, le programme permet de sélectionner les places désirées par le client et un ticket est émis pour chaque place réservée.

Il est demandé de faire le diagramme de classes de cette application.
Ici, nous profiterons de ce même énoncé pour prolonger la solution et approfondir sur ce cas concret le problème soulevé par la mise en correspondance entre l'analyse objet et la base de données relationnelle dans laquelle sont stockées en dernier recours toutes les informations. Nous proposerons une manière très simple et sans doute honteusement bricolée d'affronter la difficulté posée par cette mise en correspondance. Nous ne prétendons nullement à l'excellence ici, et la solution proposée en est une parmi beaucoup d'autres, témoignant de l'absence à ce jour d'une méthodologie unifiée et unanime pour affronter ce type de difficulté.

La première étape pourrait consister en la réalisation de la base de données et de son schéma relationnel comme représenté sur la figure 19-7.

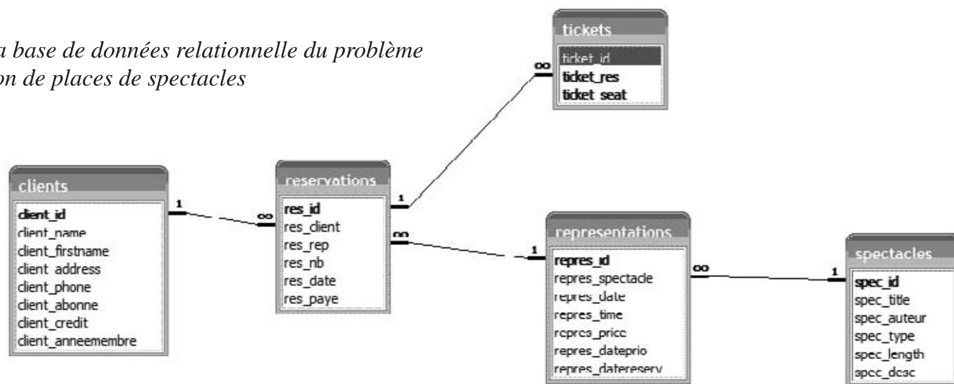


Figure 19-7
Schéma de la base de données relationnelle du problème de réservation de places de spectacles

On y découvre les cinq tables auxquelles on est en droit de s'attendre : `Reservation`, `Spectacle`, `Representation`, `Client` et `Ticket`. Remarquez d'office que les deux types de client, selon qu'ils soient abonnés ou non, ne se distinguent ici que par l'attribution d'un attribut booléen client abonné dans la table `Client`. L'héritage n'existe pas en relationnel. En revanche, dans le diagramme de classe de la même application

représenté dans la figure 19-8, on retrouve bien les deux sous-classes de `Client`, les abonnés et les non-abonnés. La classe `Client` devient dès lors une classe abstraite, tout comme les méthodes de réservation et de paiement.

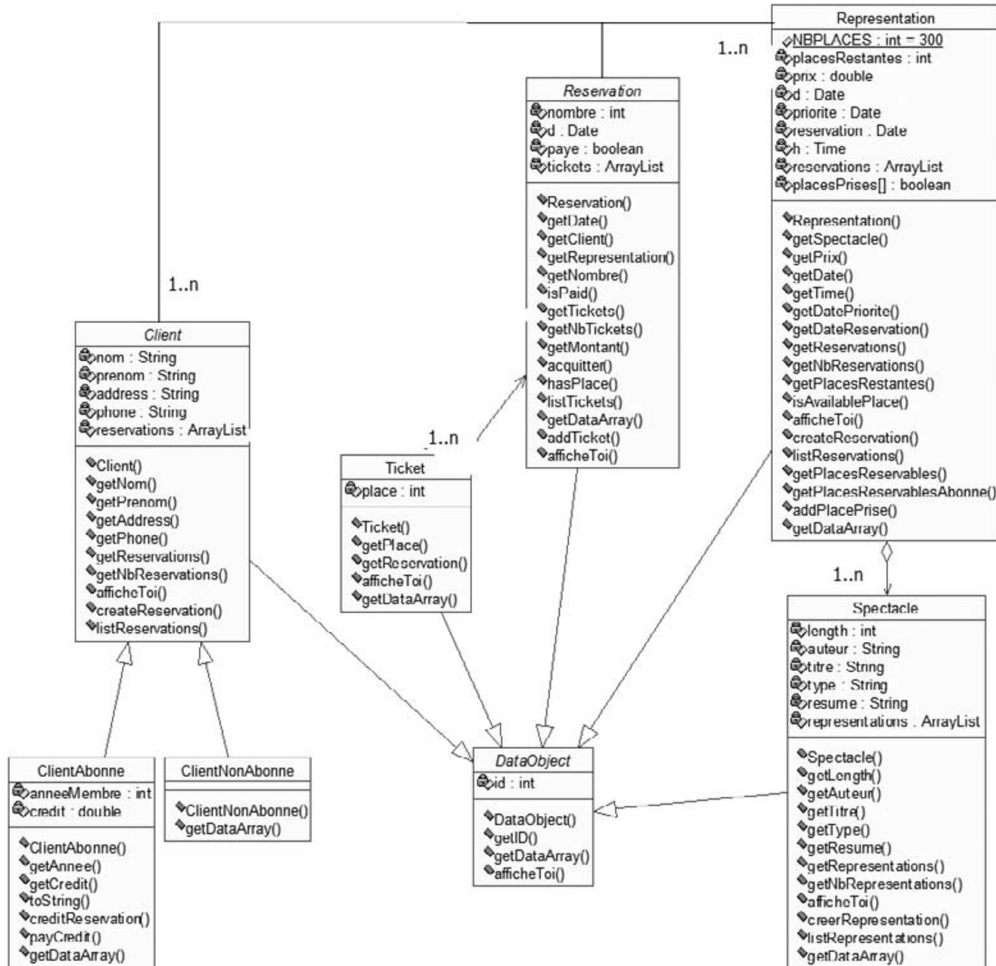


Figure 19-8

Diagramme de classe UML du problème de réservation de places de spectacles

L'addition de la classe DataObject doit vous paraître encore plus intriguante. Voici son code Java :

```
public abstract class DataObject {
      private int id;
      public DataObject(int id){
            this.id = id;
      }
```

```
        public int getID(){
            return this.id;
        }
        public abstract Object[] getDataArray();
        public abstract String afficheToi();
        public String toString(){
            return this.afficheToi();
        }
}
```

C'est une des premières conséquences de la mise en correspondance entre ces deux réalités informatiques que sont l'objet et le relationnel. Le seul attribut de cette classe est l'entier `id` qui, en fait, reprend la valeur de la clé primaire de tous les objets figurant également dans la base de données relationnelle. C'est la raison pour laquelle les classes `Client`, `Spectacle`, `Representation`, `Reservation` et `Ticket` héritent toutes de cette superclasse. Chaque objet de notre code reprendra les attributs de son enregistrement correspondant y compris un attribut additionel en la personne de la clé primaire. La redéfinition de la méthode `getDataArray()` dans la sous-classe client abonné sera par exemple :

```
public Object[] getDataArray() {
            Object[] array = new Object[Client.HEADERS.length];
            array[0] = new Integer(this.getID());
            array[1] = this.getNom();
            array[2] = this.getPrenom();
            array[3] = this.getAddress();
            array[4] = this.getPhone();
            array[5] = "Oui";
            array[6] = new Double(this.getCredit());
            array[7] = new Integer(this.getAnnee());
            array[8] = new Integer(this.getNbReservations());
            return array;
    }
```

Tous les attributs du client sont entreposés dans un tableau d'objets que l'on présentera dans une fenêtre lorsqu'il faudra afficher toutes les informations concernant le client. Vous pouvez également découvrir dans le diagramme de classe les différentes méthodes associées aux classes importantes du projet. Par exemple, le client abonné pourra : créer une réservation, afficher ses réservations, payer ses réservations, créditer une réservation, payer son crédit, etc. La classe `Reservation` est indiquée dans le diagramme de classe comme une classe d'association entre les classes client et representation. Les liaisons `1-n` de la base de données et du diagramme de classe obligent la construction de l'objet du côté `n` à s'associer systématiquement à l'objet du côté `1`. Ainsi, la construction d'un objet issu de la classe d'association `Reservation` se fera comme suit :

```
public Reservation(int id, Date d, Client client, Representation representation, int nb, boolean paye){
            super(id);
            this.d = d;
            this.client = client;
            this.representation = representation;
            this.nombre = nb;
            this.paye = paye;
            this.tickets = new ArrayList();
            this.representation.addReservation(this);
            this.client.addReservation(this);
    }
```

Tant la `representation` que le `client` rajoutent cette réservation, en instance de création, dans leur array list de réservations respectives. Dans la solution présentée ici, l'accès à la base de données relationnelle, se fait au tout début de l'exécution. Ainsi, par exemple, l'extraction des clients et des spectacles de la base de données relationnelle se fera comme suit :

```
public void loadData(){
            this.extractClients();
            this.extractSpectacles();
            this.extractRepresentations();
            this.extractReservations();
            this.extractTickets();
      }
public ArrayList extractClients(){
            clients = new ArrayList();
            try {
              if(conn==null)
                    this.connect();
                  Statement stmt = conn.createStatement();
                  String query = "SELECT * FROM clients ORDER BY client_name, client_firstname";
                  ResultSet rs = stmt.executeQuery(query);
                  while(rs.next()){
                        int c_id = rs.getInt("client_id");
                        String c_n = rs.getString("client_name");
                        String c_fn = rs.getString("client_firstname");
                        String c_ad = rs.getString("client_address");
                        String c_ph = rs.getString("client_phone");
                        int c_ab = rs.getInt("client_abonne");
                        double c_c = rs.getDouble("client_credit");
                        int c_am = rs.getInt("client_anneemembre");
                        if(c_ab==1){
                              clients.add(new ClientAbonne(c_id,c_n,c_fn,c_ad,c_ph,c_c,c_am));
                        } else {
                              clients.add(new ClientNonAbonne(c_id,c_n,c_fn,c_ad,c_ph));
                        }
                  }
                  rs.close();
                  stmt.close();
            } catch (SQLException e) {
                  e.printStackTrace();
            } catch(Exception e){
                  e.printStackTrace();
            }
            return clients;
      }
      public ArrayList extractSpectacles(){
            spectacles = new ArrayList();
            try {
              if(conn==null)
                    this.connect();
                  Statement stmt = conn.createStatement();
                  String query = "SELECT * FROM spectacles ORDER BY
```

```
                spec_auteur, spec_title";
                ResultSet rs = stmt.executeQuery(query);
                while(rs.next()){
                      int s_id = rs.getInt("spec_id");
                      String s_t = rs.getString("spec_title");
                      String s_a = rs.getString("spec_auteur");
                      String s_ty = rs.getString("spec_type");
                      int s_l = rs.getInt("spec_length");
                      String s_r = rs.getString("spec_desc");
                      spectacles.add(new
                      Spectacle(s_id,s_l,s_a,s_t,s_ty,s_r));
                }
                rs.close();
                stmt.close();
          } catch (SQLException e) {
                e.printStackTrace();
          } catch(Exception e){
                e.printStackTrace();
          }
          return spectacles;
    }
```

L'extraction des réservations de la base de données relationnelle afin de constituer l'array list de réservations est quelque peu plus délicate vu son positionnement comme classe d'association entre les deux classes précédentes. Il faut donc, afin de retrouver le client et la réprésentation associée à cette réservation en particulier, faire explicitement l'intersection entre les clés primaires des clients et des représentations et leur clé étrangère respective présente dans la classe `Reservation`. C'est la raison des deux boucles et des tests dans le code qui suit :

```
public ArrayList extractReservations(){
                if(representations.isEmpty())
                this.extractRepresentations();
          if(clients.isEmpty())
                this.extractClients();
          reservations = new ArrayList();
          try {
            if(conn==null)
                  this.connect();
                Statement stmt = conn.createStatement();
                String query = "SELECT * FROM reservations ORDER BY
                res_client, res_rep, res_date";
                ResultSet rs = stmt.executeQuery(query);
                while(rs.next()){
                      int r_id = rs.getInt("res_id");
                      int r_c = rs.getInt("res_client");
                      int r_r = rs.getInt("res_rep");
                      int r_n = rs.getInt("res_nb");
                      Date r_d = rs.getDate("res_date");
                      boolean r_p = rs.getBoolean("res_paye");
                      Client c = null;
                      for(int i=0;i<clients.size();i++){
```

```
                        c = (Client) clients.get(i);
                        if(c.getID()==r_c)
                              break;
                  }
                  Representation rp = null;
                  for(int i=0;i<representations.size();i++){
                        rp = (Representation) representations.get(i);
                        if(rp.getID()==r_r)
                              break;
                  }
                   reservations.add(new
                   Reservation(r_id,r_d,c,rp,r_n,r_p));
            }
            rs.close();
            stmt.close();
      } catch (SQLException e) {
            e.printStackTrace();
      } catch(Exception e){
            e.printStackTrace();
      }
      return reservations;
}
```

Pour finaliser l’application, il reste encore à réaliser une interface graphique qui sera capable une fois les clients et les représentations affichées d’en sélectionner l’un et l’autre afin d’effectuer la réservation et le paiement qui la validera. Mais il se fait tard et c’est ici que nous vous abandonnons dans le développement de cette application…

Les bases de données relationnelles-objet

Nous avons vu qu’alors que le stockage de données reste la chasse gardée de la technologie des bases de données relationnelles, l’orientation objet, en revanche, s’est répandue comme une traînée de poudre dans l’ensemble des applications logicielles. Le fossé entre ces deux pratiques n’est pas tant d’ordre sémantique, toutes deux privilégiant une mise en relation entre les types de données qu’elles représentent, mais plutôt d’ordre « mécanique », la concrétisation de ces relations s’opérant de façon très différente, le relationnel détaché de tout emplacement physique, l’OO attaché à une installation physique des objets et à leur utilisation par adressage explicite. De multiples tentatives sont à l’œuvre depuis quinze ans, afin de combler ce fossé et d’éviter, lors du développement des logiciels, une telle dénaturation du modèle objet dès la sauvegarde de ceux-ci.

Permettant l’interfaçage avec ces bases de données, le langage SQL s’est fait, par ses différentes évolutions, le meilleur témoin de ces tentatives. On peut, en substance, considérer que ces essais de conciliation proviennent des deux bords. En premier lieu, on a affaire aux défenseurs à tous crins du relationnel pour le stockage, qui proposent un enrichissement de SQL appelé SQL3, parfaitement compatible avec le premier, et offrant aux développeurs d’applications OO une manière de stocker leur objet plus en accord avec la manière de les utiliser.

En second lieu, on a affaire aux défenseurs à tous crins de l’OO pour le développement logiciel, qui considèrent que la sauvegarde sur disque dur se devrait de simplement « miroiter » la RAM à un moment donné, sans se

détacher aucunement de la pratique OO, et qui proposent un langage d'interfaçage avec ces bases de données nouvelle génération, nouveau lui aussi, OQL, inspiré de SQL mais ne maintenant plus la compatibilité.

Chris J. Date et le manifeste pour les bases de données du futur

Chris Date est devenu ces dernières années le porte-étendard du modèle relationnel, surtout dans la formalisation logique qui, non seulement l'accompagne, mais l'a également précédé comme en témoignent les écrits de Codd, le créateur et théoricien d'origine du modèle relationnel. L'ouvrage de Date, *An Introduction to Database Systems* chez Addison-Wesley, est la référence essentielle pour tous ceux qui veulent découvrir le modèle relationnel à travers une « lorgnette » plus formelle. S'il va sans dire que cette théorisation n'est pas toujours requise pour une utilisation, se satisfaisant souvent de l'intuition, des logiciels informatiques de base de données relationnelle, un détour du côté de la théorie permet souvent de conforter cette approche, en prenant conscience de la force et de la cohérence logique de ce modèle.

Irrité par les critiques incessantes dont le modèle relationnel faisait les frais face à la popularisation croissante de l'approche OO et même d'XML ces dernières années, et reprochant au premier d'être insuffisant pour pallier la persistance des objets et la souplesse d'XML, Chris Date a, plus récemment, rédigé un ouvrage intitulé *Fondation for future Database System*, toujours chez Addison-Wesley. Cet ouvrage a pour ambition d'élargir les développements théoriques du premier, pour permettre au modèle relationnel d'intégrer les objets et ce qui les relie entre eux (comme la relation d'héritage). Il n'y a rien, selon son auteur, qui empêche la vision relationnelle d'intégrer les objets, et les solutions théoriques proposées vont, de ce fait, plutôt vers un schéma de base de données du type relationnel/objet (SQL3 étant une possible généralisation de SQL allant dans le même sens).

Cet ouvrage reste donc fortement ancré dans la tradition relationnelle et, à nouveau, sa nature essentiellement théorique pourra décourager plus d'un lecteur. Il faut clairement l'appréhender comme la suite logique du premier, dont on peut se passer dans la pratique, mais qui permet de comprendre d'où viennent la puissance et le caractère incontournable de l'approche relationnelle pour la sauvegarde des données. Force est de reconnaître qu'un peu d'ordre théorique ne fait pas de mal pour sortir de la situation confuse dans laquelle la communauté informatique se trouve, dès qu'il faut stocker ses objets sur le disque dur.

Selon l'auteur : « *The relational model needs no extension, no correction, no subsumption – and above all, no pervesion ! – in order for it to support those few desirable features that are commonly regarded as aspects of object-orientation, because they are orthogonal to the relational model.* » Si la position définitive de Chris Date, en faveur d'une vision essentiellement relationnelle de la persistance des objets, ne mettra pas un terme aux nombreuses controverses que suscite la question (en raison des enjeux économiques qui sont liés au secteur des bases de données), elle a le mérite d'insister sur une nécessaire distinction que nous faisons dans ce chapitre. Il est capital, dans le problème du stockage de données, de séparer les aspects implémentations physiques, fortement marqués par l'approche OO, des aspects de nature logique des données et de leur relation, qui font la force du relationnel.

L'essentiel des difficultés créées par les tentatives de réconciliation entre l'OO et le relationnel vient de la confusion entre ces deux visions. Et séparation il y a, car que la proie et le prédateur soient conceptuellement liés, cela reste vrai où qu'ils vivent, où qu'ils meurent et quel que soit le lieu de leur sauvegarde. Ainsi, le système de clés, de jonction entre les valeurs et de maintien de l'intégrité référentielle semble une voie conceptuellement irremplaçable, car physiquement indépendante pour la mise en relation des objets. La question centrale, au centre des débats, est donc celle du maintien et de la viabilité de cette séparation entre le support physique et la nature logique des données.

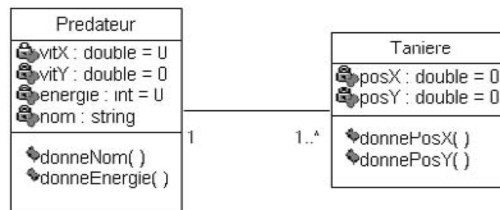
Commençons par les premiers, ardents partisans du stockage relationnel, et proposant SQL3 comme possible compromis au choc culturel dressant, l'un contre l'autre, le relationnel et l'OO. Oracle en est un des représentants les plus actifs, pour ne pas dire combatifs, et on le comprend aisément, vu l'omniprésence de cet acteur industriel dans le monde des bases de données relationnelles. Une version actuelle de cette base de données, (on doit en être à la dixième version aujourd'hui), est compatible avec SQL3, et offre aux programmeurs OO, surtout Java en l'occurrence, la possibilité d'intégrer des requêtes SQL3 dans la partie JDBC de leur code.

Les programmeurs ne sont plus contraints à des « acrobaties syntaxiques » infernales pour réaliser la sauvegarde de leurs objets. L'OO est installé de manière progressive, tout en maintenant l'existant relationnel comme tel. Malheureusement, SQL3 n'est pas réputé pour sa simplicité comme nous allons le voir, dans l'obligation qu'il est de rester compatible avec les versions précédentes de SQL.

SQL3

Nous nous bornerons, à propos des prédateurs et de leurs tanières, à présenter quelques-unes des requêtes permises par ce nouvel SQL, « relooké objet ». Tout d'abord, SQL3 introduit, en plus des tables, afin d'obtenir un équivalent des classes, plusieurs nouveaux types de données : les « types de données abstraits », les « types références » et les « types collections ». Reprenons le diagramme UML `Predateur-Taniere`, et ébauchons en SQL3 la création des types abstraits correspondants.

Figure 19-9
Le diagramme de classe UML des classes prédateur et tanière.




```
CREATE TYPE PredateurType AS OBJECT

CREATE TYPE TaniereType AS OBJECT
( posX NUMBER,
  posy NUMBER,
  lePredateur REF PredateurType )

CREATE TYPE TaniereRefType AS OBJECT
(taniereRef REF TaniereType)

CREATE TYPE tableRefTaniereType AS TABLE OF TaniereRefType

CREATE OR REPLACE TYPE PredateurType AS OBJECT
( vitX NUMBER,
  vitY NUMBER,
  energie NUMBER,
  nom  VARCHAR(20),
  lesTanieres tableRefTaniereType )

CREATE TABLE Predateur OF PredateurType NESTED TABLE lesTanieres STORE AS tableLesTanieres

CREATE TABLE Taniere OF TaniereType
```

On peut considérer que ce code SQL3 est pour la base de données sous-jacente la contrepartie de la création des classes pour le programme. Si nous le décortiquons ligne par ligne, on trouve tout d'abord la création du premier type (`Predateur`) sans rien (mais que nous définirons complètement dans la suite) simplement parce que nous l'utilisons dans la définition des tanières (et ce, pour solutionner le problème des références croisées). Toute création de table se doit d'abord d'être précédée du type « abstrait ». Dans la définition du type « abstrait » pour les tanières, on retrouve bien un attribut référentiel du type « prédateur ».

On s'aperçoit, en effet, du rapprochement permis par SQL3 avec la démarche objet, car, à l'instar de l'OO, la relation entre les deux types abstraits s'établit par l'intermédiaire d'attributs référentiels. On crée un type référentiel sur les tanières qu'on appelle `TaniereRefType`. Ensuite, on crée un vecteur de ces référents appelé `tableRefTaniereType`. Si on avait, à ce stade, voulu créer un tableau de taille finie, on aurait plutôt écrit l'instruction suivante : `CREATE TYPE tableRefTaniereType AS VARRAY(5) OF TaniereRefType`.

On définit ensuite le type `PredateurType` que l'on avait juste introduit pour solutionner la référence croisée. On voit qu'il contient comme attribut un référent vers `tableRefTaniereType`, ce qui concrétise la partie `1..*` de l'association UML. Après avoir créé les types abstraits, on retrouve maintenant la création des tables, propre à SQL en général, mais qui s'effectue, cette fois, comme « instance » des types abstraits. Vu la présence du référent de type « vecteur », l'instanciation de la table `Predateur` requiert la déclaration d'une table enchâssée (NESTED TABLE). Reconnaissez qu'en matière de simplicité, c'est un sommet.

L'héritage est également pris en compte dans SQL3 et, en se souvenant qu'un prédateur est une sous-classe de `Faune`, on pourrait déclarer le type abstrait prédateur comme ceci :

```
CREATE TYPE PredateurType UNDER FauneType
```

De manière à se coller davantage encore à l'OO, SQL3 permet également l'installation de méthodes, qui peuvent retourner quelque chose ou non (fonction ou procédure), à l'intérieur de ces types abstraits. Ainsi, considérons la méthode d'accès `donnePosX()` de la tanière, retournant la valeur du premier attribut. Sa prise en compte se réalisera de cette manière :

```
CREATE OR REPLACE TYPE TaniereType AS OBJECT
( posX NUMBER,
  posy NUMBER,
  lePredateur REF PredateurType
  MEMBER FUNCTION donnePosX RETURN NUMBER )

CREATE TYPE BODY TaniereType AS
  MEMBER FUNCTION donnePosX RETURN NUMBER IS
  BEGIN
    RETURN posX;
  END donnePosX;
END;
```

Enfin, on maintient dans SQL3 l'équivalent des requêtes classiques `SELECT` ou `INSERT`.

```
SELECT t.posX, t.lePredateur.nom
FROM Taniere t
WHERE t.lePredateur.energie = 100
INSERT INTO Predateur
VALUES (PredateurType(1,2, …))
```

Dans le `Select`, on retrouve l'utilisation du « . » qui permet d'accéder aux attributs de l'objet. On constate bien l'apport de la pratique OO dans la syntaxe SQL classique. Dans une approche purement relationnelle, il y aurait lieu d'effectuer dans ce type de requête une jointure explicite entre les tables `Predateur` et `Taniere`. Ce nouvel SQL permet aux programmeurs OO une écriture plus légère, et qui ne surprendra plus outre mesure par des manipulations tortueuses et des détournements syntaxiques. C'est un petit pas vers l'OO, non négligeable, mais encore timide vu la pression exercée par le relationnel et la syntaxe SQL existante.

Les bases de données orientées objet

Toutefois, en tant que programmeur OO, quelle serait la solution idéale ? Elle consisterait, très simplement, en un mode d'écriture et de lecture d'objets aussi trivial que la sérialisation, mais qui emprunte aux bases de données toutes leurs indispensables qualités fonctionnelles telles que les archivages automatisés, les accès sécurisés, fiabilisés, concurrentiels, l'existence d'un langage d'interrogation détaché de la programmation équivalent à SQL mais pleinement objet cette fois, tous les outils de visualisation et d'analyse intégrés, on en passe et des meilleures... Une telle perspective existe et a pour nom : les bases de données orientées objets. Elles existent, nous les avons rencontrées. Elles ont quitté le monde du fantasme pour celui des disques durs, et de nombreux produits sont, depuis 15 ans, apparus sur le marché, tels que ObjectStore, Versant, O2, Poet, Gemstone, entre autres.

Bien que ces produits s'emploient à réussir le mariage idéal entre la sérialisation et les qualités fonctionnelles du relationnel, leur percée commerciale est encore timide et le restera sans doute longtemps. La raison en est évidente. Allez dire à tous ces utilisateurs de bases de données relationnelles, qui, depuis tant d'années, stockent des quantités invraisemblables de données dans leurs tables, de transformer ce mode de stockage en un mode OO. Vous serez bien reçu.

Alors que les informations resteront encore, pour longtemps, bien au chaud, dans des tables qui sont en relations, nul entrepreneur ne ressentant le besoin de les transformer en objets, même la définition plus formalisée de ce que seraient ces bases de données version objet fait encore l'objet de quelques controverses. En dépit de nombreuses tentatives, les bases de données OO sont encore loin d'une possible standardisation, comme l'ont été UML, CORBA ou SQL.

Néanmoins, un consortium d'acteurs du monde industriel des bases de données s'est créé depuis douze ans, l'ODMG (Object Data Management Group), qui tente de parvenir à l'établissement d'un standard pour les bases de données OO. Ce consortium lié de très près à l'OMG n'arrête pas de se défaire et de se reconstituer sous des formes très diverses, autre illustration de la difficulté toujours pas résolue de réconcilier l'objet et le relationnel. Parmi leurs cibles, un nouveau langage d'interfaçage avec ces bases, dénommé OQL (Object Query Langage), est en passe de se stabiliser… De s'imposer ? C'est une autre histoire.

OQL

Ce langage peut être utilisé de façon isolée ou, plus naturellement, en le plongeant dans du code objet (Java, C++…). OQL se doit donc d'être extrêmement proche des langages OO. Ainsi, pour la création des classes, on retrouve des instructions très familières comme :

```
class Predateur : Faune {
  attribute int energie ;
  attribute string nome ;
  relationship Taniere lesTanieres inverse lePredateur;
  string donneNom();
  int donneEnergie();
}
```

Et pour la création d'un objet :

```
  Declare p :Predateur
    p = Predateur(nom : « Gates »,…..) // le constructeur
```

OQL permet d'écrire des requêtes de type `SELECT`, mais pas d'`UPDATE`, car les méthodes sont là pour pourvoir aux mises à jour. Ces requêtes appliquées à un objet, une collection d'objets ou de valeurs donnent comme

résultat une valeur, un objet, une collection de valeurs ou une collection d'objets. En voici quelques échantillons :

```
SELECT p.nom
FROM p IN Predateur
Where p.donneEnergie() > 100
```

Le résultat sera une collection de string. On découvre une écriture très proche de l'OO où l'appel des méthodes est parfaitement imbriqué dans les requêtes.

```
SELECT STRUCT (nom : p.nom, energie: p.energie)
FROM p IN Predateur

SELECT t.posX, t.posY
FROM t in Taniere, p in t.lePredateur
WHERE p.nom = "Gates"
```

Le résultat en sera un ensemble de valeurs structurées contenant chacune un nom et un entier, dans le premier cas, et deux entiers dans le second. Il y a également moyen de regrouper les résultats par une requête comme :

```
SELECT p
FROM p IN Predateur
GROUP BY (faible: p.energie < 50, fort: p.energie > 100)
```

On imagine volontiers qu'une convergence sollicitée et enviée par les développeurs OO serait celle de SQL3 et OQL, leur permettant de sauvegarder et d'interroger leurs objets d'une manière unique, quelle que soit la façon définitive dont ces derniers s'installent dans le disque dur, à la sauce relationnelle ou à la sauce OO. Cette convergence est-elle, en effet, possible, nonobstant l'existence de ces deux mécanismes de sauvegarde ? Il semble en tous les cas qu'elle reste encore hors de portée, vu la distance énorme séparant les lieux d'origine de ces deux solutions techniques, SQL d'un côté, les langages OO de l'autre.

Objets sains et saufs

Il existe aujourd'hui de multiples solutions pour permettre aux objets de maintenir leur état entre deux itérations d'un programme qui les concerne. Pour un ensemble de raisons faciles à justifier, liées à la facilité d'accès et à la transparence de la sauvegarde, et surtout permettant de préserver leur structure d'objets et la nature de leurs liens avec les autres objets, les bases de données OO semblent une solution de choix. En attendant celles-ci, et de manière à interfacer les données stockées aujourd'hui dans les bases de données relationnelles avec les codes OO, une initiative comme SQL3 est tout aussi vitale. Pourra-t-on un jour manipuler indifféremment, à l'intérieur d'un programme OO, enregistrements de bases de données et objets stockés à même le disque ? L'avenir nous le dira.

Linq

La bibliothèque Linq (Langage-INtegrated Query), sous les feux de la rampe informatique à l'heure actuelle, est sans conteste l'innovation majeure de la nouvelle version de la plate-forme de développement de Microsoft .Net3. On peut aller jusqu'à penser que l'essentiel des autres innovations présentes dans cette troisième version, telles les expressions lambda, les « méthodes d'extension », les variables anonymes typées implicitement « var » et la manipulation des « query », sont autant d'étapes majeures pour enfin aboutir à la réalisation de Linq. Si Linq est à ce point discuté et décortiqué dans le monde informatique, c'est bien que

cette bibliothèque est présentée par Microsoft comme la solution à cet épineux problème encore irrésolu de la mise en correspondance entre le monde du relationnel et de l'orienté objet, auquel vient s'ajouter depuis quelques années la nécessaire prise en charge de l'information stockée et représentée sous forme XML.

Une même information, et trois manières de la représenter : l'objet, le relationnel et XML. Il est grand temps de parvenir à homogénéiser un mode d'accès et de traitement de cette information qui soit indépendant de la manière de la représenter. Qu'importe que l'hôtel que je cherche sur le Web soit enregistré dans une base de données relationelles, orientée objet ou sous forme XML. Il m'est urgent de savoir au plus vite la disponibilité des chambres et le prix. Linq se présente comme un candidat possible à cette homogénéisation : une même requête circulant dans ces trois univers d'information.

De surcroît, Linq s'intègre parfaitement aux langages de programmation .Net tels C# et VB, il en est une évolution. Toute requête Linq se trouve compilée au même titre que n'importe quelle autre instruction. Fini donc les requêtes SQL qui ne posent problème qu'à l'exécution, c'est-à-dire quand, en effet, elles interagissent directement avec la base de données relationnelle. Linq permet de prévenir par la compilation des erreurs d'interrogation de bases de données (par exemple des attributs mal orthographiés) qui, auparavant, ne se seraient produites que pendant la phase d'exécution. De plus, l'homogénéisation permettra à une même expression d'interroger indifféremment des systèmes d'information objet, XML et relationnels et d'en mélanger les résultats.

Mais voyons directement, afin d'en savourer toute la richesse, deux exemples de code contenant chacun une requête Linq d'un même type `Select` mais s'appliquant, pour la première, sur un tableau d'objets (ici des `strings`) et pour la deuxième, sur notre habituelle base de données relationnelles des prédateurs.

Premier exemple de Linq agissant sur un tableau d'objets

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

class app {
       static void Main() {
           string[] names = {"Burke","Connor","Frank","Everet","Albert","George",
           "Harris","David"};

           IEnumerable<String> test = from s in names
                                      where s.Length == 5
                                      orderby s
                                      select s.ToUpper();

           foreach (string item in test) Console.WriteLine(item);
       }
   }
```

Résultat

```
ALAIN
DAVID
MARCO
```

Deuxième exemple de Linq agissant sur une base de données relationnelles

```
using System;
using System.Linq;
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Data.OleDb;

[Table] public class Predateur
{
    [Column(IsPrimaryKey=true)] public int idPredateur;
    [Column]      public int vitx;
    [Column]      public int vity;
    [Column]      public int energie;
}

class SomeDataContext : DataContext

{
     public SomeDataContext(IDbConnection connection) : base (connection) {}

}

class Test
{
    static void Main() {
         string strConnection = "Provider=Microsoft.Jet.OleDb.4.0;";
         strConnection += "Data Source=C:\\Test\\Ecosysteme.mdb";
         OleDbConnection connexion = new OleDbConnection(strConnection);
         SomeDataContext dataContext = new SomeDataContext(connexion);
         Table<Predateur> lesPredateurs= dataContext.GetTable<Predateur>();

         IQueryable<int> query = from p in lesPredateurs
             where p.energie > 0
             orderby p.idPredateur descending
             select p.idPredateur;

         foreach (int id in query) Console.WriteLine (id);
    }
}
```

Résultat

```
2
1
```

Dans la base de données, seuls les deuxième et troisième prédateurs possèdent une énergie différente de 0, mais les deux `id` sont affichés en ordre inverse en conséquence de la clause `orderby`.

On découvre qu'un même type de `select`, très inspiré de SQL mais inversé, composé d'un même filtre de type `where` et/ou d'un mécanisme de tri `orderby`, peut s'appliquer indifféremment sur un tableau d'objets

entreposés dans la mémoire RAM et sur une base de données relationnelles. Une troisième illustration aurait pu être possible, en appliquant, une fois encore, ce même type de requête à même l’information stockée en XML.

C’est ce qu’illustre la figure 19-10 schématisant la bibliothèque Linq.

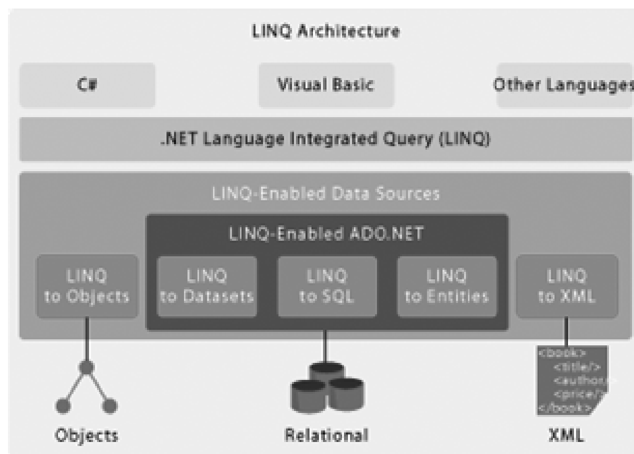


Figure 19-10
La bibliothèque Linq.

Si, par mégarde, il y a erreur dans la requête `select` – attribut mal orthographié par exemple, le code produira une erreur à la compilation. Cela est évidemment possible grâce à la déclaration de la table|classe `Predateur`, qui fait le lien avec la base de données relationnelles.

Afin de parvenir à Linq, il a fallu d’abord accepter des instructions de la forme :

```
IEnumerable<string> test1 = Enumerable.Where(names, s=> s.Length == 5);
```

dans lesquelles, le prédicat booléen du `Where` est entré en argument et appliquée sur la liste `names`. En fait, cette écriture est transformée en celle qui suit :

```
IEnumerable<string> f = names.Where(s=> s.Length == 5);
```

On peut voir que la fonction `Where` dans laquelle le prédicat booléen est toujours passé en argument est maintenant directement appliqué sur la liste de string `names`. La notion de variables anonymes `var` est une addition également nécessaire qui permet de créer le type désiré sur le fil comme dans l’instruction :

```
var test3 = from s in names
                              where s.Length == 5
                               orderby s
                               select s.ToUpper();
```

Quant à la composition des clauses `Where`, `Orderby` et d’autres possibles encore, il s’agit de fonctions qui s’exécutent de manière emboîtée, la deuxième s’exécutant sur les résultats obtenus par la première et ainsi de suite. Ces requêtes, une fois compilées et transformées, seront, dans le cas précis d’une base de données relationnelle, transformées en requêtes SQL exactes afin d’attaquer la base de données, et le résultat sera récupéré sous la forme d’une liste énumérable.

Exercices

Exercice 19.1

Décrivez les quatre solutions de persistance envisageables pour la sauvegarde des objets, entre deux itérations d'un programme les concernant.

Exercice 19.2

Expliquez pourquoi les classes de type « stream » se prêtent assez naturellement à la mise en pratique des mécanismes d'héritage.

Exercice 19.3

Expliquez le pourquoi des « flux filtrés ».

Exercice 19.4

Justifiez la dénomination de la « sérialisation ».

Exercice 19.5

Expliquez pourquoi il est nécessaire d'interrompre parfois cette sérialisation pour certaines classes d'objet.

Exercice 19.6

Différenciez classes et tables.

Exercice 19.7

Comment peut-on traduire une relation d'héritage entre deux classes en une simple relation entre deux tables ?

Exercice 19.8

Justifiez pourquoi une simple relation n-n entre deux classes ne peut se traduire en une simple relation n-n entre deux tables correspondantes.

Exercice 19.9

Justifiez la nécessité des bases de données OO.

Exercice 19.10

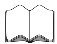
Pourquoi SQL a donné naissance à deux extensions OO différentes : SQL3 et OQL, et pourquoi l'avenir des programmeurs OO serait-il plus radieux si ces deux-là ne formait plus qu'un ?

20

Et si on faisait un petit flipper ?

Ce chapitre est dédié à la réalisation d'un petit flipper en Java. C'est sa conception orientée objet qui nous intéressera principalement ici. À quelques variations près, cette application orientée objet est empruntée à Timothy Budd, un auteur prolifique en ouvrages introductifs à l'OO de très bonnes factures, et certainement un des meilleurs éducateurs en la matière qu'il nous ait été donné de découvrir. Ce flipper nous permettra de nous interroger sur une série de choix de conception : héritage versus *composition, héritage simple* versus *multihéritage, conception décentralisée* versus *centralisation, dans un contexte plus pratique et surtout très familier. Glissez 50 cents dans la fente.*

Sommaire : Flipper — Conception orientée objet



Candidus — Et si on jouait un peu avec notre mécano… J'aimerais bien voir ce qu'on peut faire avec quelques objets graphiques tout simples !

Doctus — Bonne idée ! Et tu vas voir que c'est pas si simple de faire simple.

Cand. — Faire simple a toujours été un défi pour les informaticiens. À ma connaissance, on n'y parvient que d'une seule manière, en passant par les étapes suivantes : il faut d'abord réaliser une première version ; l'étape suivante consiste à crasher son disque dur pour n'avoir plus aucune chance de succomber à la tentation de récupérer quoi que ce soit de la première version ; à ce stade, on repart de zéro en ayant soin d'éviter de refaire les mêmes bêtises que dans la version précédente.

Doc. — Quoi ! Ça, je connais, comme tout le monde. Mais j'aurais un peu de mal à recommander une telle méthode de travail !

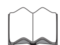
Cand. — Et comment l'OO nous permettrait-elle d'en trouver une meilleure ?

Doc. — En voici une : tu termines d'abord la première lecture de mon ouvrage et tu prends ensuite quelques jours de vacances au Tibet,. Après quelques boucles de ce genre, le nombre de questions sur l'OO qui encombrent ta cervelle aujourd'hui sera alors comparable au nombre de réponses que tu pourras donner.

Cand. — Charmant programme !

Doc. — Il s'agit d'un cheminement initiatique très gratifiant malgré tout. Chaque étape t'apportera un élément nouveau qui s'emboîtera comme par enchantement dans ton édification.

Cand. — Alors c'est dit ! Mais je vais commencer mon voyage avec une partie de flipper au bistrot du coin, histoire de voir comment ça marche…



Timothy Budd

La simulation du flipper, qui fait l'essentiel de ce chapitre, est inspirée d'un ouvrage de Timothy Budd intitulé *Introduction to Object-Oriented Programming* et publié chez Addison-Wesley. Nous voudrions, par ce petit encart, rendre à César ce qui appartient à César en même temps rendre un hommage appuyé aux qualités didactiques de cet auteur, professeur au département informatique de la Oregon State University, et passé maître dans la pédagogie des concepts OO. Ces quatre derniers ouvrages, tous publiés par Addison-Wesley, *Understanding Object-Oriented Programming with Java – 1998*, *C++ for Java Programmers – 1999*, *Classic Data Structures in Java – 2001*, *An Introduction to Object-Oriented Programming – 2002 (le livre en est à sa troisième édition et s'améliore sans cesse)*, sont de véritables petites bibles, où puisent plus d'un professeur en informatique pour leurs cours de programmation. En plus du flipper, vous y trouverez des simulations de billard, jeux de cartes ou jeux d'échecs. L'auteur a raison de consacrer aux jeux une telle place dans son enseignement de l'OO. Les jeux, plus que toute autre réalité qui nous entoure, se prêtent à une analyse de classe généralement très simple et à un développement enthousiaste de la part des apprentis programmeurs. On s'amuse plus encore en programmant un jeu qu'en y jouant.

Un premier ouvrage, paru en 1989, *A Little Smalltalk*, avait déjà ravi par l'excellence de la présentation qui était faite de Smalltalk et la mise à disposition d'un compilateur *Little Smalltalk*. Nul doute que notre ouvrage doive beaucoup aux écrits du professeur Budd tant dans les aspects techniques qui y sont développés que dans les voies pédagogiques que nous avons tenté de suivre…

En couverture de ses derniers ouvrages figure le même animal, une espèce d'ornithorynque dont la présence tient plus du gag (les informaticiens ont souvent l'habitude d'associer un animal à une innovation technologique, O'Reilly en a fait sa marque de fabrication – pour Python le choix de l'animal n'a pas posé d'énorme problème), sans rapport évident avec le contenu de l'ouvrage. Puisse notre ouvrage avoir hérité les qualités pédagogiques de cet auteur. Mais comme il le dit lui-même : « Chaque fois que j'enseigne un cours, il y a toujours un moment en lisant les notes de cours où je me fais la réflexion : "Tu sais, je pense que je pourrais écrire un bien meilleur cours que celui-ci. Par malheur, je me le dis aussi quand il s'agit de mes propres livres". ». C'est on ne peut plus fidèle à notre manière de penser également. Si quoi que ce soit vous échappe dans ce livre, n'hésitez pas à honorer votre bibliothèque de la présence d'un petit ornithorynque.

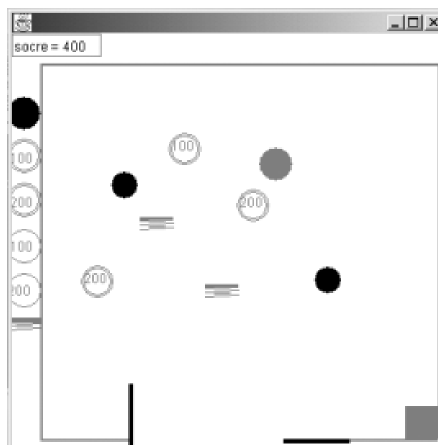


Figure 20-1

La simulation en Java du petit flipper.

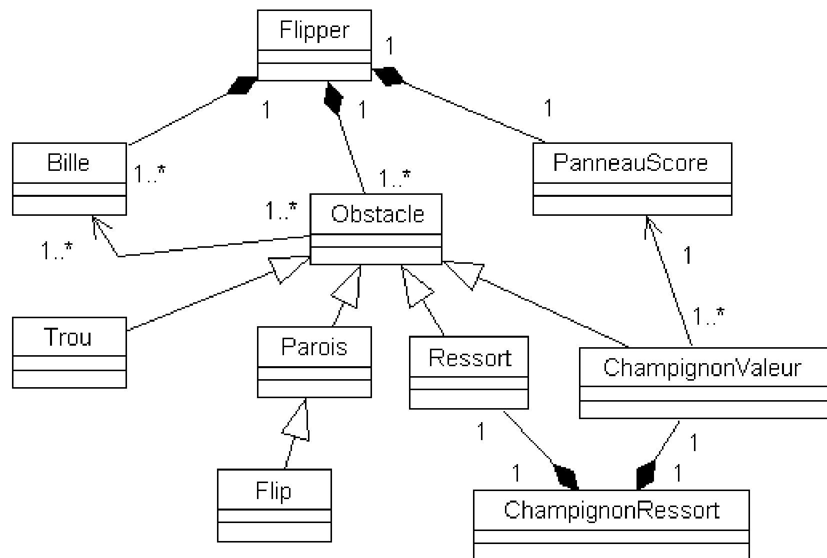
Généralités sur le flipper et les GUI

Comme vous pouvez l'observer sur la figure 20-1, le flipper est constitué d'une bille plus claire, du point de départ de la bille en bas à droite, et d'une rangée d'obstacles possibles à gauche. Dans un premier temps, il s'agit de sélectionner et d'installer quelques-uns de ces obstacles dans la partie principale du flipper, à droite. Quatre catégories d'obstacle peuvent se trouver dans le flipper, le trou (le rond noir) qui fait disparaître la bille, le ressort qui la fait juste rebondir, le champignon de valeur qui, quand il est heurté par la bille, incrémente le score de la valeur indiquée sur l'obstacle, et finalement le champignon à ressort, qui, également, incrémente le score de la valeur indiquée mais, en plus, fait rebondir la bille, tout comme le ressort.

Enfin, nous trouvons dans le flipper un panneau de score, des parois qui entourent le flipper et font rebondir la bille, et deux flips (dont un est dressé dans l'image) qui permettent de renvoyer la bille quand elle est à leur portée. La bille est lancée en cliquant avec la souris sur l'emplacement en bas à droite, sa direction et vitesse dépendant de l'endroit où on clique. Les deux flips sont contrôlés par deux touches du clavier.

Le flipper est intéressant dans l'analyse orientée objet car plusieurs points délicats de conception doivent être solutionnés avant de s'attaquer à l'écriture du code proprement dite. Ces différents cas seront discutés et comparés par l'entremise des diagrammes de classe qui les accompagnent. Le diagramme de classe final que nous avons décidé d'implémenter est présenté en figure 20-2. Par un souci de clarté, les méthodes et les attributs sont exclus de l'analyse. Seuls nous intéressent les éléments de base du jeu et la manière dont ils interagissent. Nous fournissons en fin de chapitre le code complet Java de ce flipper.

Figure 20-2
Diagramme de classe du flipper.



Nous allons passer en revue les différentes parties de ce diagramme, et les choix d'implémentation correspondants. La classe principale est la classe `Flipper`. C'est elle qui agrège tous les éléments du flipper et contient la méthode `main()` qui lance le jeu. Ce flipper est composé d'un vecteur de billes, d'un vecteur d'obstacles et d'un panneau de score. Le lien est un lien fort, un lien de composition, car il est difficile d'imaginer toutes ces

parties du jeu existant à l'extérieur du flipper. N'a-t-on jamais vu un flipper qui fuit ? Sinon, rien de bien particulier pour les billes.

Dans une version écartée de ce diagramme mais conservée dans le code Java qui suit, la bille hérite de la classe `Trou`. La raison en est (comme vous pouvez le voir dans l'image du flipper) que la bille et le trou se dessinent de la même façon, un petit disque coloré. La bille peut, dès lors, récupérer la méthode dessinant le trou, *via* un héritage.

Quand un héritage de la sorte se produit, il s'agit plutôt d'une pratique un peu tortueuse, permettant à une classe de récupérer, à moindre frais, des fonctionnalités déclarées dans une autre. C'est un héritage très « utilitariste », de convenance, mais sans légitimité sémantique, et, en conséquence, d'assez mauvais aloi. Gardez toujours en votre chef ce credo fondamental de l'OO : la réalité est le juge ultime. En effet, une bille n'est pas une espèce de trou. C'est un coup de chance s'ils se dessinent de la même manière, et cela pourrait ne plus être le cas dans des versions ultérieures du code, versions toujours à prévoir. L'utilitarisme forcené a ceci de délicat qu'il n'est pas forcément partagé par tous les praticiens.

Pourtant, les programmeurs Java sont très friands de ce type de pratique, quand ils font de leur classe, une héritière d'un « Frame » ou d'un « Thread », pour récupérer les fonctionnalités graphiques de la première et la gestion concurrentielle de la seconde. À la place, la classe pourrait, plus proprement, agréger un frame ou un thread. Cela permettrait, notamment, de ne pas gaspiller le seul héritage disponible en Java ou C#. Comme vous pouvez l'observer dans les codes suivants (Java et C#), dont le rôle est de faire apparaître sur l'écran une fenêtre comprenant un bouton qui réagit lorsqu'on clique dessus en écrivant dans la console le texte inscrit sur le bouton. Une pratique courante, tant en Java qu'en C#, est de faire hériter la classe principale d'un `JFrame` pour Java et d'une `Form` pour C#.

Code Java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class TestFenetre extends JFrame implements ActionListener {
 private JButton unBouton;

 public TestFenetre() {
  unBouton = new JButton("un tres gros bouton");  /* on crée le bouton */
  unBouton.addActionListener(this);               /* on crée une sensibilité à l'évènement sur le
bouton, en y associant la méthode actionPerformed */
  setTitle("Essai Fenetre");                      /* hérité de JFrame, on intitule la fenêtre */
  setSize(400, 400);                              /* on la dimensionne */
  setLocation(0,0);                               /* on la positionne */
  getContentPane().add(unBouton, "North");        /* on ajoute le bouton sur la fenêtre au-dessus */
  setVisible(true);                               /* le tout doit apparaître */
 }

 public void actionPerformed (ActionEvent e) {     /* on définit ce qui se passe en clickant sur le
bouton , cette méthode provient de l'interface ActionListener et est à rédéfinir obligatoirement */
      System.out.println(e.getActionCommand());   /* on récupère le texte sur le bouton */
 }
```

```
 public static void main(String[] args) {
  new TestFenetre();
 }
}
```

Résultat du code Java

Figure 20-3
Création en Java d'une fenêtre avec un large bouton.

Aussi déroutante que puisse paraître, à première vue, la création d'un objet à même sa classe, l'instruction `new TestFenetre()` dans la méthode `main` permet de créer l'objet `JFrame` que l'on voit apparaître. Dans le constructeur de la classe principale, ce sont, pour la plupart, les méthodes héritées du `JFrame` que l'on appelle, méthodes servant à intituler, dimensionner et positionner le `JFrame`. Puis on « additionne » le bouton sur le `JFrame`. Voici la version quasi équivalente de ce code en C# :

Code C#

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class TestFenetre : Form {
 private Button unBouton;
 public TestFenetre() {
  unBouton        = new Button();                    /* on crée le bouton */
  unBouton.Text   = "un tres gros bouton";           /* on le nomme */
  unBouton.Click += new EventHandler (boutonMethode); /* on y ajoute par délégué la sensibilité au
click */
  Text            = "Essai Fenetre";                 /* on nomme la fenêtre */
  StartPosition   = FormStartPosition.CenterScreen;  /* on positionne la fenêtre */
  ClientSize      = new Size(400, 400);              /* on la dimensionne */
  unBouton.Dock = DockStyle.Top;                   /* on place le bouton au-dessus */
  Controls.Add(unBouton);                            /* on ajoute le bouton */
 }
```

```
 public void boutonMethode(object envoyeur, EventArgs e) {
              Console.WriteLine(((Button)envoyeur).Text); /*la concrétisation du délégué récupère
   le  texte du bouton */
 }

 public static void Main() {
  Application.Run(new TestFenetre()); // on fait apparaître la fenêtre
 }
}
```

Dans le code C#, l’action du clic sur le bouton se fait par concrétisation du délégué plutôt que par redéfinition de la méthode provenant de l’interface comme en Java, mais l’esprit est très semblable. L’instruction `Application.Run(new TestFenetre());` a pour mission de faire apparaître la fenêtre.

Dans ces deux langages, une version moins répandue et pourtant plus souhaitable, substituant à l’héritage la composition, respectant en cela l’héritage pour ce qu’il est, est donnée ci-après (hormis la gestion du bouton non prise en compte ici, le résultat produit sera exactement le même) :

En Java

```
import java.awt.*;
import javax.swing.*;
public class Fenetre2 {
  JFrame uneFenetre;
  JButton unBouton;

  public Fenetre2() {
    uneFenetre  = new JFrame();
    unBouton    = new JButton("Un très gros
    ➥bouton");
    uneFenetre.setTitle("Essai Fenetre");
    uneFenetre.setSize(400, 400);
    uneFenetre.setLocation(0,0);
    uneFenetre.getContentPane().add(unBouton,
    ➥"North");
    uneFenetre.setVisible(true);
  }
  public static void main(String[] args) {
    new Fenetre2();
  }
}
```

En C#

```
using System;
using System.Windows.Forms;
using System.Drawing;
public class Fenetre2 {
  private Button unBouton;
  private static Form uneFenetre;
```

```
    public Fenetre2() {
      unBouton       = new Button();
      uneFenetre     = new Form();
      unBouton.Text  = "un tres gros bouton";
      uneFenetre.Text = "Essai Fenetre";
      uneFenetre.StartPosition = FormStartPosition
      ➥.CenterScreen;
      uneFenetre.ClientSize = new Size(400, 400);
      uneFenetre.Controls.Add(unBouton);
      Application.Run(uneFenetre);
    }
    public static void Main() {
      new Fenetre2();
    }
  }
```

Dans ces deux codes, l’héritage a été remplacé par un mécanisme de composition entre les classes. Nos deux classes principales possèdent une fenêtre plutôt qu’elles n’en héritent. Nous avons vu au chapitre 11 la mise en œuvre très semblable, surtout pour les attributs, du mécanisme d’héritage et de composition. Dans la plupart des cas où un mécanisme d’héritage utilitariste et sans « légitimité sémantique » est utilisé, il sera souhaitable de lui substituer une relation de composition entre les classes impliquées.

En Python

```
from Tkinter import *
class TestFenetre:

    def __init__(self,master):
      master.title("Essai Fenetre") #on intitule la fenetre
      # on la dimensionne
      frame = Canvas(master, height=400, width=400, background = "grey")
      # on définit ce qui se passe en appuyant sur le bouton
      def actionPerformed(event):
        print event.widget.cget("text")

      unBouton = Button(master,text="un tres gros bouton")
      unBouton.place(anchor="nw") # on place le bouton au nord ouest
      unBouton.bind('<Button-1>',actionPerformed) #on clique sur le bouton gauche


root = Tk() #on lance l'application graphique
unTest = TestFenetre(root)
root.mainloop()
```

La bibliothèque graphique la plus courante avec Python est celle de `Tkinter`, une adaptation de la bibliothèque Tk développée à l’origine pour le langage Tcl et repris depuis par Perl. Notez qu’il est toujours possible via l’interface `Jython` de récupérer les librairies graphiques de Java. Une fois importé, le module Tkinter est très simple d’emploi. Créer, configurer et positionner les widgets souhaités ne présente aucune difficulté. Après avoir créé et placé les widgets, le code appelle la fonction `mainloop` de Tkinter qui initie une boucle principale amenant le code à ne plus être piloté que par les événements. Le module Tkinter fournit un certain

nombre de widgets très simples qui couvrent l'essentiel des besoins de base de toute application GUI : `Button`, `Checkbutton`, `Entry`, `Label`, `ListBox`, `RadioButton`...

Une petite animation en C#

Comme un avant-goût du flipper Java, une petite entrée en matière en sorte, nous présentons une très simple animation codée en C#, dont le résultat à l'exécution est représentée sur la figure 20-4. Il s'agit de quatre parois entre lesquelles circule une série de billes. Pour lancer une bille, il suffit de cliquer sur le bouton du haut. Chaque bille se déplace entre les parois et rebondit sur celles-ci. Le diagramme de classe des trois classes principales est également représenté. On y voit l'interaction entre toutes les billes et les quatre parois. Le code exploite une classe présente dans les librairies .Net, la même que l'on retrouvera dans le code Java du flipper. Il s'agit de la classe `Rectangle`, qui permet de gérer simplement les intersections entre les objets, ici entre les billes et les parois. En effet, cette classe possède en son sein une méthode `IntersectsWith(Rectangle)` qui renvoie `true` lorsque le rectangle en question croise celui qui lui est passé en argument. Il suffit dès lors de positionner, comme les codes à venir le font, tant les billes que les parois dans des objets de la classe `Rectangle` pour que l'intersection entre les parois et les billes devienne un jeu d'enfant, à l'instar du flipper. C'est une parfaite illustration de la réutilisation d'une classe prévue à cet effet dans les librairies accompagnant les langages de programmation. Chaque bille change de couleur lorsqu'elle rencontre une paroi. On s'aperçoit aussi dans le code de l'utilisation d'un multithreading, indispensable pour toute animation graphique. Chaque bille possède son propre thread qui est créé à la création de la bille (le thread est donc un « composite » de la bille) et s'occupe du déplacement de cette dernière. Ce bout de code devrait vous permettre de mieux appréhender celui du flipper qui nous verrons par la suite et qui est plus complet.

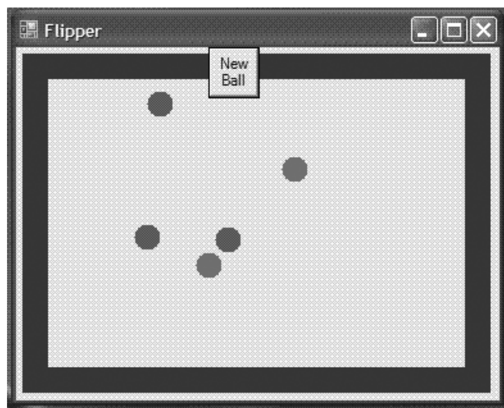


Figure 20-4

Petite animation graphique réalisée en C# dans laquelle une série de billes se balade en rebondissant sur les parois.

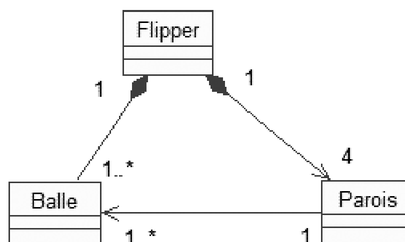


Figure 20-5

Diagramme de classe de cette application

Code C# de l'animation graphique

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Threading;

public class Paroi
{
       private Rectangle r;
       private Color c;
       private ArrayList lesBalles;

       public Paroi(int x, int y, int width, int height, Color c, ArrayList lesBalles)
       {
              r = new Rectangle(x,y,width,height);
              this.c = c;
              this.lesBalles = lesBalles;
       }

       public void interagitBalle() // interaction entre les parois et la balle
       {
              foreach (Balle b in lesBalles)
              {
                     if (r.IntersectsWith(b.getR))
                     {
                           b.changeDirection(r.Width>r.Height?'Y':'X');
                           b.changeColor();
                     }
              }
       }

       public void dessine(Graphics g)
       {
              g.FillRectangle(new SolidBrush(c),r.X,r.Y,r.Width,r.Height);
       }
}

public class Balle
{
       private Rectangle r; // le rectangle pour gérer l'intersection
       private Color c;
       private int vitx, vity;
       private Thread t; // chaque balle a son thread
       private Flipper f;

       public Balle(int x, int y, int vitx, int vity,
              Color c, Flipper f)
       {
              r = new Rectangle(x,y,20,20);
              this.c = c;
              this.vitx = vitx;
              this.vity = vity;
```

```
        this.f = f;
        t = new Thread(new ThreadStart(tBalle));
        t.Start();
}

public void changeColor()
{
        if (c==Color.Red)
        {
                c = Color.Green;
        } else
        {
                c = Color.Red;
        }
}

public Rectangle getR
{
        set
        {
        }
        get
        {
                return r;
        }
}

public void changeDirection(char axe)
{
        if (axe == 'X')
        {
                vitx = -vitx;
                r.X += 5*vitx;
        }
        else
        {
                vity = -vity;
                r.Y += 5*vity;
        }
}

public void bouge()
{
        r.X+=vitx;
        r.Y+=vity;
        for (int i = 0; i<4; i++)
        {
          f.lesParois[i].interagitBalle();
        }
}

public void dessine(Graphics g)
{
        g.FillEllipse(new SolidBrush(c),r.X,r.Y,
                r.Width, r.Height);
}
```

```
        public void tBalle()
        {
                while (true)
                {
                        Thread.Sleep(1);
                        bouge();
                        f.Invalidate();
                }
        }
}

public class Flipper:Form
{
        public Paroi[] lesParois = new Paroi[4];
        private ArrayList lesBalles = new ArrayList();
        private Random ro = new Random();

        public Flipper()
        {
                Init();
        }

        private void Init()
        {
                Size = new Size(385, 310);
                this.Text = "Flipper";

 /* Les trois instructions incompréhensibles suivantes permettent de régler
 les problèmes de scintillement ou de clignotement de l'écran */

                SetStyle(ControlStyles.DoubleBuffer, true);
                SetStyle(ControlStyles.UserPaint,true);
                SetStyle(ControlStyles.AllPaintingInWmPaint,true);

                lesParois[0] = new Paroi(5,5,20, 250,Color.Blue,lesBalles);
                lesParois[1] = new Paroi(5,5,350,20,Color.Blue,lesBalles);
                lesParois[2] = new Paroi(5,250,350,20,Color.Blue,lesBalles);
                lesParois[3] = new Paroi(350,5,20,265,Color.Blue,lesBalles);

                Button unBouton = new Button();
                unBouton.Text = "New Ball";
                unBouton.Size = new Size(40,40);
                unBouton.Location = new Point(150,0);
                unBouton.Click += new EventHandler(Handler);
                this.Controls.Add(unBouton);

        }

        protected override void OnPaint(PaintEventArgs e) // pour dessiner l'animation
        {
                Graphics g = e.Graphics;
                for (int i=0; i<4; i++)
                {
                        lesParois[i].dessine(g);
```

```
            }
            foreach (Balle b in lesBalles)
            {
                  b.dessine(g);
            }
      }

      public void Handler(object o, EventArgs e)
      {
            lesBalles.Add(new Balle(ro.Next(150,200),ro.Next(150,200),1,1,Color.Red,this));
      }

      public static void Main()
      {
            Application.Run(new Flipper());
      }
}
```

Retour au Flipper

La classe `Obstacle` est une superclasse, dont héritent toutes les catégories d'obstacles présentées plus haut. On conçoit aisément qu'un ensemble d'attributs et de méthodes soit commun à tous les obstacles : attribut de position, de couleur, méthodes de déplacement (pour les amener sur la partie principale du flipper), etc. Les obstacles interagiront avec la bille par un lien fort d'association ou plus faible de dépendance (si la bille est passée comme argument des méthodes).

Lorsque la bille les heurte, les obstacles doivent envoyer un message à la bille, message dont la signature sera commune, `laBouleMeCogne()`, par exemple, mais dont les effets dépendront en dernier ressort de la nature de l'obstacle. En conséquence de quoi, la classe `Obstacle` est la classe abstraite idéale, intégrant une méthode `laBouleMeCogne()` abstraite. On nage en plein polymorphisme.

Toutefois, avant de consacrer davantage d'attention à ce polymorphisme, re-considérons un instant l'existence de ce lien d'association qui unit les obstacles aux billes. Pourrait-on faire sans ? Affirmatif, car le flipper sait tout ce qu'il est nécessaire de savoir, pour faire interagir les billes et les obstacles. En effet, il possède tous les référents vers les billes et les obstacles, qui lui permettent de réaliser et de gérer l'intersection entre ces deux types d'éléments.

Si rien ne l'interdit vraiment, si le diagramme de classe s'en trouverait même allégé (car cela permet de supprimer l'association entre les billes et les obstacles), et si nous avons dit que la pratique consistant à minimiser les interactions entre les classes (dans un souci de stabilité) est à encourager, il n'en reste pas moins qu'il s'agirait d'une conception OO maladroite. Avant tout, ce serait trahir la réalité dont on s'inspire, car, dans cette réalité, les obstacles ont bien une interaction directe avec la bille, et, en OO, se détourner du réel, vous l'aurez compris, ça finit toujours par se payer cash.

Si, en réalité, la réalité a toujours le dernier mot, il faut bien que les conséquences soient tout aussi effectives dans une perspective plus ingénieriste. Supposez que vous cherchiez à récupérer la classe `Obstacle` dans une application tout à fait différente, plutôt qu'un flipper, un billard, par exemple. Si le flipper s'était chargé de l'interaction entre les billes et les obstacles, l'obstacle, sorti du flipper, serait incapable de s'occuper d'une telle interaction. Sa fonction principale, sa raison d'être, sa nature intime d'obstacle, lui serait détroussée au profit du flipper, et notre pauvre obstacle se trouverait bien démuni une fois placé dans le billard. Modularité,

stabilité, évolutivité et ré-utilisabilité, sont les mots-clés de la pratique OO, contre lesquels s'inscrirait, en effet, la simple disparition du lien unissant les billes aux obstacles.

Plutôt que des obstacles vers les billes, le lien d'association pourrait être dirigé des billes vers les obstacles, ou même dans les deux sens. Là encore, ces autres versions possibles, tout aussi programmables que les deux précédentes, sont à déconseiller, car il est bon que les obstacles, selon leur nature, intègrent en leur sein leur gestion particulière de la bille. En revanche, la simple détection du choc entre l'obstacle et la bille pourrait être tout aussi bien codée dans la bille que dans les obstacles. Dans notre code Java, la partie de code reproduisant l'interaction entre la bille et les obstacles est reprise ci-après :

```
public void run () {
  while (laBoule.y() < LeFlipper.hauteurDuFlipper) {
    laBoule.deplaceToi();
    for (int j=0; j<lesObstacles.size(); j++) {
      unObstacle = (ObstacleDuFlipper)lesObstacles.elementAt(j);
      if (unObstacle.croiseLaBoule(laBoule))
        unObstacle.laBouleMeCogne(laBoule);
    }
    repaint();
    try {
      sleep(100);
    } catch (InterruptedException e) { System.exit(0); }
  }
}
```

On constate que chaque obstacle possède deux méthodes d'interaction avec la bille (ici, c'est un lien de dépendance qui réunit les obstacles et les billes, car la bille est passée en argument de l'obstacle), la détection de l'intersection, héritée de la classe obstacle, et la gestion du choc, à redéfinir de manière polymorphique pour chaque obstacle. Le même message de gestion du choc est envoyé à tous les éléments du vecteur d'obstacles, message dont l'exécution dépendra du type d'obstacle particulier.

Chaque catégorie d'obstacle est une sous-classe de la classe `Obstacle`. L'obstacle `ChampignonValeur` se particularise en ceci que cette classe interagit avec la classe `PanneauScore`. L'effet du message `laBouleMeCogne` pour cette classe-là consiste, de fait, à envoyer un message d'incrémentation de valeur au panneau de score. Les parois du flipper sont autant d'obstacles, comme les flips. Ces derniers réagissent semblablement aux parois, par un simple rebond, et donc héritent de celles-ci. Le ressort ne réagit pas exactement comme les parois car sa forme change lors du choc.

Nous allons nous intéresser plus spécifiquement maintenant à la classe `ChampignonRessort` qui, à la fois, fait rebondir la bille et incrémente le panneau de score. Il y a deux manières de relier cette classe aux deux classes dont elle partage les caractéristiques : le `ChampignonValeur` et le `Ressort`. La première, sans doute la plus naturelle, serait de recourir au multihéritage, car on peut dire, sans ambages, que le `ChampignonRessort` est, tout à la fois, un `ChampignonValeur` et un `Ressort`. Malheureusement pour les praticiens du Java, du C#, ou du PHP 5, seuls C++ et Python autoriseraient une pareille solution. En C++, si la classe hérite des deux autres, la méthode `laBouleMeCogne` du `ChampignonRessort` s'écrirait à peu près comme suit :

```
  public void laBouleMeCogne(BouleDeFlipper uneBoule) {
    ChampignonValeur ::laBouleMeCogne(uneBoule);
    Ressort ::laBouleMeCogne (uneBoule);
  }
```

On appelle, l’une après l’autre, les deux méthodes `laBouleMeCogne(uneBoule)` héritées des deux superclasses. Cela étant impossible en Java, C# et PHP 5, il reste la solution qui consiste à dire qu’un `ChampignonRessort` est composé, cette fois, et d’un `ChampignonValeur` et d’un `Ressort`. Ce faisant, nous restons fidèles à la réalité. Ici, comme souvent, la pratique d’héritage et de composition présente la même légitimité. Elles se valent parfaitement. Dans notre code, le constructeur de la classe ainsi que la méthode `laBouleMeCogne()` sont définis comme suit :

```
class ChampignonRessort extends ObstacleDuFlipper {
  private ChampignonDeValeur cdv;
  private Ressort rs;

  public ChampignonRessort(int x, int y, int v, PanneauDeScore unPanneau) {
    cdv = new ChampignonDeValeur(x,y,v,unPanneau);
    rs = new Ressort(x,y);
  }
  public void laBouleMeCogne(BouleDeFlipper uneBoule) {
    cdv.laBouleMeCogne(uneBoule);
    rs.laBouleMeCogne(uneBoule);
  }
}
```

Nous proposons le code Java qui suit pour le Flipper, très largement inspiré de celui de Timothy Budd. La classe `obstacle` a été remplacée par une interface mais l’idée est essentiellement la même. Les versions C# et Python devraient se déduire assez facilement à partir de celle de Java.

Code Java du Flipper

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.applet.*;
import javax.swing.*;

class PanneauDeScore extends Label {
 private int score;

 PanneauDeScore() {
  setText ("score = " + score);
  }

 public void incrementerScore (int v) {
  score += v;
  setText ("score = " + score);
 }

 public void paint (Graphics g) {
  g.setColor(Color.magenta);
  g.drawRect (0,0,80,20);
 }
}
```

```
class BouleDeFlipper {
 private Rectangle region;
 private double dx,dy;
 protected Color laCouleur;
 private double effetGravite;

 public BouleDeFlipper(int x, int y, int r) {
  region = new Rectangle (x-r,y-r,2*r,2*r);
  dx = -(400 - x)/5;
  dy = -(400 + y)/40;
  laCouleur = Color.blue;
  effetGravite = 0.3;
 }

 public void changeLaCouleur(Color nouvelleCouleur) {
  laCouleur = nouvelleCouleur;
 }

 public void changeLaVitesse (double ndx, double ndy) {
  dx = ndx;
  dy = ndy;
 }

 public int leRayon () {
  return (int)region.width/2;
 }

 public int x () {
  return region.x + leRayon();
 }

 public int y () {
  return region.y + leRayon();
 }

 public double getDX() {
  return dx;
 }

 public double getDY() {
  return dy;
 }

 public Rectangle getRegion() {
  return region;
 }

 public void deplaceToiEnXY (int x, int y) {
  region.setLocation (x,y);
 }
```

```
 public void deplaceToi () {
  dy = dy + effetGravite;
  region.translate ((int)dx, (int)dy);
 }

 public void dessineToi (Graphics g) {
  g.setColor (laCouleur);
  g.fillOval (region.x, region.y, region.width, region.height);
 }
}

interface ObstacleDuFlipper {
 public boolean croiseLaBoule (BouleDeFlipper uneBoule);
 public void deplaceToiEn (int x, int y);
 public void dessineToi (Graphics g);
 public void laBouleMeCogne(BouleDeFlipper uneBoule);
}

class Ressort implements ObstacleDuFlipper {
 private Rectangle plateau;
 private int etat;

 public Ressort(int x, int y) {
  plateau = new Rectangle (x,y,30,3);
  etat = 1;
 }

 public void deplaceToiEn (int x, int y) {
  plateau.setLocation (x,y);
 }

 public void dessineToi(Graphics g) {
  int x = plateau.x;
  int y = plateau.y;
  g.setColor (Color.green);

  if (etat==1) { // ressort comprimé
   g.fillRect(x,y,plateau.width, plateau.height);
   g.drawLine (x, y+3, x+30, y+5);
   g.drawLine(x+30, y+5, x, y+7);
   g.drawLine(x, y+7, x+30, y+9);
   g.drawLine(x+30, y+9, x, y+11);
  }
  else { // ressort détendu
   g.fillRect(x, y-8, plateau.width, plateau.height);
   g.drawLine(x, y+5, x+30, y-1);
   g.drawLine(x+30, y-1,x,y+3);
   g.drawLine(x,y+3,x+30,y+7);
   g.drawLine(x+30,y+7,x,y+11);
   etat = 1;
  }
 }
```

```
 public boolean croiseLaBoule (BouleDeFlipper uneBoule) {
  return plateau.intersects(uneBoule.getRegion());
 }

 public void laBouleMeCogne(BouleDeFlipper uneBoule) {
  if (uneBoule.getDY() > 0) {
   uneBoule.changeLaVitesse (uneBoule.getDX(), -uneBoule.getDY());
  }

  uneBoule.changeLaVitesse (uneBoule.getDX(), uneBoule.getDY() - 0.5);
  etat = 2;
 }
}

class Parois implements ObstacleDuFlipper {
 protected Rectangle region;
 private Color maCouleur;

  public Parois (int x, int y, int width, int height) {
   region = new Rectangle (x, y, width, height);
   maCouleur = Color.yellow;
  }

  public void setCouleur(Color maCouleur) {
   this.maCouleur = maCouleur;
  }

  public void deplaceToiEn (int x, int y) {
   region.setLocation (x,y);
  }

  public void dessineToi (Graphics g) {
   g.setColor (maCouleur);
   g.fillRect(region.x, region.y, region.width, region.height);
  }

  public boolean croiseLaBoule (BouleDeFlipper uneBoule) {
   return region.intersects(uneBoule.getRegion());
  }

  public void laBouleMeCogne(BouleDeFlipper uneBoule) {
   if (region.width > region.height) {
    uneBoule.changeLaVitesse(uneBoule.getDX(),
                -uneBoule.getDY());
   }
   else {
    uneBoule.changeLaVitesse(-uneBoule.getDX(),
                uneBoule.getDY());
   }
  }
 }

 class Flip extends Parois {
  private int fx,fy;
```

```
  private boolean pressed;

  public Flip(int x, int y, int width, int height, int fx, int fy) {
   super(x,y,width, height);
   setCouleur(Color.black);
   this.fx = fx;
   this.fy = fy;
   pressed = false;
  }

  public void bouge() {
  if (! pressed)
  {
   int a,b;
   a = fx;
   b = fy;
   fx = region.x;
   fy = region.y;
   region.x = a;
   region.y = b;
   a = region.width;
   b = region.height;
   region.width = b;
   region.height = a;
   pressed = true;
  }
 }

  public void release() {
   if (pressed)
   {
   int a,b;
   a = fx;
   b = fy;
   fx = region.x;
   fy = region.y;
   region.x = a;
   region.y = b;
   a = region.width;
   b = region.height;
   region.width = b;
   region.height = a;
   pressed = false;
  }
 }
}

class Trou extends BouleDeFlipper implements ObstacleDuFlipper {
 public Trou (int x, int y) {
  super (x, y, 12);
  changeLaCouleur(Color.black);
 }
```

```
 public void deplaceToiEn (int x, int y) {
  super.deplaceToiEnXY (x,y);
 }

 public boolean croiseLaBoule(BouleDeFlipper uneBoule) {
  return getRegion().intersects(uneBoule.getRegion());
 }

 public void laBouleMeCogne(BouleDeFlipper uneBoule) {
  uneBoule.deplaceToiEnXY (0, LeFlipper.hauteurDuFlipper + 30);
  uneBoule.changeLaVitesse(0,0);
 }
}


class ChampignonDeValeur extends Trou {
 private int valeur;
 private PanneauDeScore lePanneau;

 public ChampignonDeValeur(int x, int y, int valeur, PanneauDeScore lePanneau) {
  super(x,y);
  this.valeur = valeur;
  this.lePanneau = lePanneau;
  changeLaCouleur(Color.red);
 }

 public void laBouleMeCogne(BouleDeFlipper uneBoule) {
  lePanneau.incrementerScore (valeur);
 }

 public void dessineToi(Graphics g) {
  g.setColor(laCouleur);
  g.drawOval (getRegion().x, getRegion().y, getRegion().width, getRegion().height);
  String s = "" + valeur;
  g.drawString(s,getRegion().x, y() + 2);
 }
}

class ChampignonRessort implements ObstacleDuFlipper {
 private int etat;
 private ChampignonDeValeur cdv;
 private Ressort rs;

 public ChampignonRessort(int x, int y, int v, PanneauDeScore unPanneau) {
  cdv = new ChampignonDeValeur(x,y,v,unPanneau);
  rs = new Ressort(x,y);
  etat = 1;
 }

 public boolean croiseLaBoule(BouleDeFlipper uneBoule) {
  return cdv.getRegion().intersects(uneBoule.getRegion());
 }
```

```
 public void deplaceToiEn (int x, int y) {
  cdv.deplaceToiEn(x,y);
  rs.deplaceToiEn(x,y);
 }


 public void dessineToi(Graphics g)  {
  cdv.dessineToi(g);
  if (etat == 2) { // le rond est en extension
   g.drawOval(cdv.getRegion().x-6,cdv.getRegion().y-2,
      cdv.getRegion().width+8,cdv.getRegion().height+8);
   etat = 1;
  }
  else {
   g.drawOval(cdv.getRegion().x-2,cdv.getRegion().y-2,
      cdv.getRegion().width+4,cdv.getRegion().height+4);
  }
 }

 public void laBouleMeCogne(BouleDeFlipper uneBoule) {
  cdv.laBouleMeCogne(uneBoule);
  rs.laBouleMeCogne(uneBoule);
  while (croiseLaBoule(uneBoule))
   uneBoule.deplaceToi();
  etat = 2;
 }
}


public class LeFlipper extends Frame implements KeyListener {
 private PanneauDeScore unPanneauDeScore;
 private Vector lesObstacles;
 private ObstacleDuFlipper unObstacleEnPlus;

 public static final int largeurDuFlipper = 400;
 public static final int hauteurDuFlipper = 400;
 public Vector lesBoules;

 public LeFlipper () {
  class LaSouris extends MouseAdapter {
   public void mousePressed(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();

    if ((x > largeurDuFlipper - 40) && (y > hauteurDuFlipper - 40)) {
     BouleDeFlipper nouvelleBoule = new BouleDeFlipper (x,y,15);
     lesBoules.addElement(nouvelleBoule);
     Thread nouveauThread = new FlipperThread (nouvelleBoule);
     nouveauThread.start();
    }


    if (x<40) {
     switch (y/40)
```

```
    {
     case 2: unObstacleEnPlus = new Trou(0,0); break;
     case 3: unObstacleEnPlus = new ChampignonRessort
                        (0,0,100,unPanneauDeScore); break;
     case 4: unObstacleEnPlus = new ChampignonRessort
                        (0,0,200,unPanneauDeScore); break;
     case 5: unObstacleEnPlus = new ChampignonDeValeur(0,0,100,
                        unPanneauDeScore); break;
     case 6: unObstacleEnPlus =  new ChampignonDeValeur(0,0,200,
                        unPanneauDeScore); break;
     case 7: unObstacleEnPlus = new Ressort(0,0); break;
     case 8: unObstacleEnPlus = new Parois(0,0,2,15); break;
    }
   }
}

public void mouseReleased(MouseEvent e) {
 int x = e.getX();
 int y = e.getY();

 if ((unObstacleEnPlus != null) && (x > 50) && (y < hauteurDuFlipper - 40)) {
  unObstacleEnPlus.deplaceToiEn(x,y);
  lesObstacles.addElement(unObstacleEnPlus);
  repaint();
 }
}}


LaSouris laSouris = new LaSouris();
addMouseListener (laSouris);


setSize (largeurDuFlipper, hauteurDuFlipper);

lesBoules = new Vector();

System.out.println("add OK");
addWindowListener (new WindowAdapter()
   {public void windowClosing (WindowEvent e)
    {
     System.exit(0);
    }
   });

 addKeyListener(this);


 lesObstacles = new Vector();
 lesObstacles.addElement(new Parois(30, 50, 2, 340));
 lesObstacles.addElement(new Parois(30, 50, 360, 2));
 lesObstacles.addElement(new Parois(390, 50, 2, 340));
 lesObstacles.addElement(new Parois(30,390,80,2));
 lesObstacles.addElement(new Parois(280,390,110,2));
 lesObstacles.addElement(new Flip(110,390,60,4, 110, 340));
 lesObstacles.addElement(new Flip(250,390,60,4, 280, 340));
```

```
    unPanneauDeScore = new PanneauDeScore();
    add("North",unPanneauDeScore);
    unPanneauDeScore.addKeyListener(this);
}

 public void keyPressed(KeyEvent evt) {
   int c = evt.getKeyCode();
   switch (c) {
    case KeyEvent.VK_LEFT :
     ((Flip)lesObstacles.elementAt(5)).bouge();
     repaint();
     break;
    case KeyEvent.VK_RIGHT :
     ((Flip)lesObstacles.elementAt(6)).bouge();
     repaint();
     break;
   }
 }

  public void keyReleased(KeyEvent evt) {
    int c = evt.getKeyCode();
    switch (c) {
    case KeyEvent.VK_LEFT :
     ((Flip)lesObstacles.elementAt(5)).release();
     repaint();
     break;
    case KeyEvent.VK_RIGHT :
     ((Flip)lesObstacles.elementAt(6)).release();
     repaint();
     break;
   }
  }

  public void keyTyped(KeyEvent evt) {
  }


 public static void main(String[] args) {
  LeFlipper leMonde = new LeFlipper();
  leMonde.addKeyListener(leMonde);
  leMonde.setVisible(true);
 }

public class FlipperThread extends Thread {
 private BouleDeFlipper laBoule;

 public FlipperThread (BouleDeFlipper laBoule) {
  this.laBoule = laBoule;
 }

 public void run () {
  while (laBoule.y() < LeFlipper.hauteurDuFlipper)
  {
```

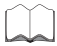
```
    laBoule.deplaceToi();
    for (int j=0; j<lesObstacles.size(); j++)
    {
     ObstacleDuFlipper unObstacle = (ObstacleDuFlipper)lesObstacles.elementAt(j);
     if (unObstacle.croiseLaBoule(laBoule))
     {
      unObstacle.laBouleMeCogne(laBoule);
     }
    }
    repaint();
    try {sleep(100);
      } catch (InterruptedException e) {System.exit(0);}
    }
   }
  }

  public void paint(Graphics g) {
   g.setColor (Color.yellow);
   g.fillRect (LeFlipper.largeurDuFlipper-40, LeFlipper.hauteurDuFlipper-40, 30, 30);
   g.setColor (Color.red);
   g.fillOval(LeFlipper.largeurDuFlipper-40, LeFlipper.hauteurDuFlipper-40, 30, 30);
   g.setColor (Color.black);
   g.fillOval(0,80,30,30);
   g.setColor(Color.red);
   g.drawOval (2,120,26,26);
   g.drawOval(0,118,30,30);
   g.drawString("100",2,140);
   g.drawOval (2,160,26,26);
   g.drawOval(0,158,30,30);
   g.drawString("200",2,180);
   g.drawOval(0,200,30,30);
   g.drawString("100",2,220);
   g.drawOval(0,240,30,30);
   g.drawString("200",0,260);
   g.setColor(Color.green);
   g.fillRect(0,280,30,5);
   g.drawLine(0,283,30,285);
   g.drawLine(30,285,0,287);
   g.drawLine(0,287,30,289);
   g.drawLine(30,289,0,291);

   for (int i=0; i< lesBoules.size(); i++) {
    ((BouleDeFlipper)lesBoules.elementAt(i)).dessineToi(g);
   }

   for (int i=0; i<lesObstacles.size(); i++) {
    ((ObstacleDuFlipper)lesObstacles.elementAt(i)).dessineToi(g);
   }
  }
 }
```

21

Les graphes

Ce chapitre explique la manière de coder une liste liée et un graphe d'objets, en se basant sur les réseaux de neurones. Il nous permet également d'aborder la généricité en C++ et en Java.

Sommaire : Omniprésence des graphes — Liste liée — La généricité en C++ et en Java — Graphe — Conception de graphe par listes liées ou de nature récursive



Candidus. — Je pense à une chose dont nous n'avons pas encore parlé : les tableaux de données. L'OO propose-t-elle des solutions particulières d'organisation à ce sujet ?

Doctus. — Elle a même inauguré tout un travail de définition des outils de manipulation des ensembles de données, sous la forme de collections d'informations pouvant être stockées, supprimées, ordonnées, parcourues, reliées, etc.

Cand. — Je pourrai donc potasser chacune des classes de collections quand j'en aurai besoin. Mais par exemple, quels devraient être mes choix pour représenter un groupe d'ordinateurs ?

Doc. — Il ne s'agit pas seulement d'en faire une simple liste de noms de machines. Nous aurons intérêt à prévoir d'associer à chacune la liste des autres machines auxquelles elle est connectée. Nous sommes en fait en train de réaliser un graphe.

Cand. — Ta démarche consiste donc à identifier la nature de notre collection…

Doc. — Exactement. Dans notre optique qui se veut aussi proche que possible d'une expression naturelle, l'architecture d'un système se construit à partir de composants dont il nous faut absolument connaître les caractéristiques.

Cand. — Dans notre cas, nous aurons donc besoin d'une liste d'éléments liés les uns aux autres.

Doc. — Pour organiser des collections d'objets, tu seras souvent amené à les ordonner. Comment ferais-tu ?

Cand. — Je les comparerais deux à deux et pour ce faire, je développerais une méthode de comparaison entre instances.

Doc. — Voilà ! Et cela nous amène à une commodité du C++, celle de la *généricité*, qui vise à faciliter l'adaptation de ce genre de méthodes pour différentes classes d'objet.

Cand. — C'est vrai après tout, les fonctions de comparaison se ressemblent énormément. On veut toujours savoir lequel de deux éléments d'un certain type est plus grand que l'autre…

Doc. — D'où l'existence des *templates*, ces sortes de modèles de méthodes, dont le type d'objet reste à déterminer ! C'est le compilateur qui en tire ensuite une version adaptée à chaque classe en substituant le « joker » indéterminé par le nom de la classe concernée.

Cand. — Cette fois, c’est du code-source qu’on met en facteur ! Et ça n’existe que dans C++ ?

Doc. — Pas tout à fait : sachant que toute classe hérite de la superclasse Objet, on pourrait créer des méthodes universelles pour peu qu’elles ne manipulent que des instances de la classe Objet.

Cand. — Oh là là ! J’imagine un jongleur à qui on lancerait trois arguments : `Jongleur.play(une balle, un serpent à sonnette, un autobus)` !

Doc. — Oui, la solution n’est pas sans risque !

Le monde regorge de réseaux

Le monde regorge d’objets simples dans leur structure et leur comportement, mais connectés entre eux. Pensez d’abord à la chimie et ses réseaux atomiques que sont les molécules, ou à ses réseaux de molécules chimiques connectées entre elles par des réactions ; pensez ensuite à la biologie et à ses nombreux réseaux : écosystèmes, réseau de neurones, réseau immunitaires, réseaux génétiques ; pensez également à la sociologie, l’économie, la politique, et partout vous serez confrontés à des myriades d’agents ou de simples objets, connectés entre eux par une structure de graphe.

Un graphe n’est rien d’autre, comme montré dans la figure ci-après, qu’un ensemble de nœuds connectés deux à deux par des arêtes. Nous ne nous intéresserons ici qu’aux graphes dits « non directionnels », et pour lesquels le sens de l’arrête est sans importance. Un tel graphe sera également dit symétrique. Un nœud est relié à un autre, un point c’est tout. C’est par exemple le cas d’une liaison entre deux atomes dans une molécule.

Mais ce n’est pas forcément le cas d’une synapse reliant deux neurones, dont l’intensité décrit le parcours d’un signal chimique d’un neurone à l’autre, parcours qui peut différer en fonction de son sens. C’est le cas de la distance entre des villes sur un itinéraire, mais cela pourrait, tristement, ne pas être le cas de liaisons d’amitié entre deux personnes, dont l’intensité peut varier, là encore, selon le sens considéré. C’est la faute aux graphes.

Cependant, si nous nous limitons à la seule existence de la liaison, sans se préoccuper outre mesure de son intensité, comme nous le ferons ici, la symétrie sera une propriété intrinsèque à ces graphes. De manière très générale, nous parlerons ici de nœuds interconnectés. Chaque nœud intégré dans un graphe sera relié à un ensemble d’autres nœuds, eux-mêmes également reliés à un ensemble de nœuds, et ainsi de suite, jusqu’à constituer le graphe dans son entièreté. La figure qui suit illustre un tel graphe composé de onze nœuds.

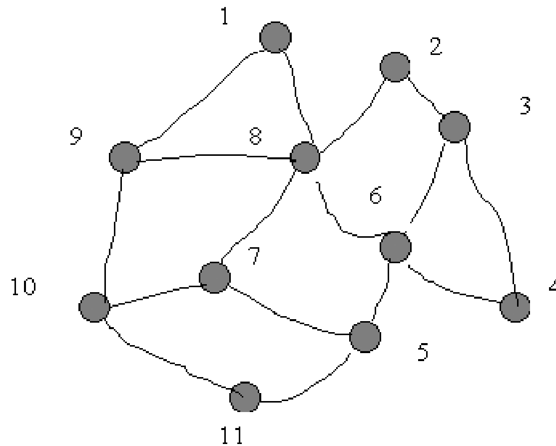


Figure 21-1
Un graphe de 11 nœuds connectés entre eux.

Dans le prochain chapitre, nous décrirons les molécules comme des graphes d'atomes. À cette fin, il importe de savoir comment, en programmation OO, on peut coder un graphe de la manière la plus simple et la plus flexible qui soit. Il nous faudra répondre à cette question en plusieurs étapes. Nous illustrerons ces étapes en nous référant aux réseaux de neurones, et plus particulièrement au réseau de Hopfield.

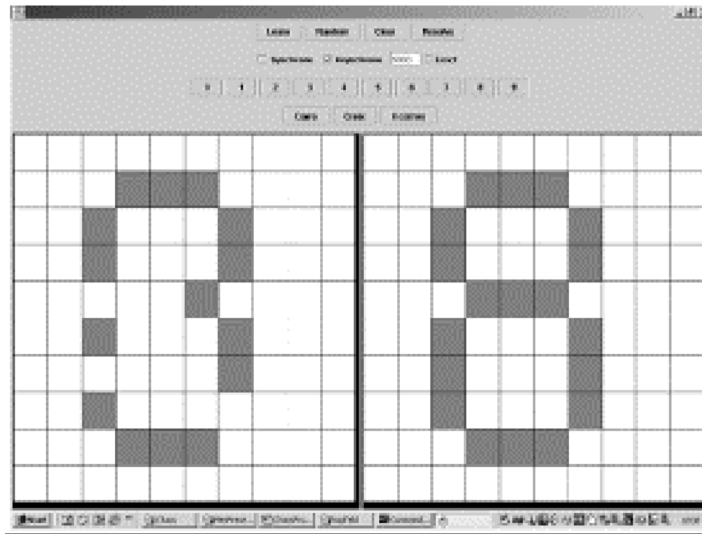


Figure 21-2

On retrouve le chiffre 8 appris à partir d'une version bruitée de ce même chiffre.

John Hopfield

John Hopfield est un chercheur de l'université de Princeton dont le nom est indéfectiblement lié à toute recherche ayant pour objet les réseaux de neurones. Il fait partie de ceux qui ont relancé cette recherche au début des années 1980, en la sortant de la léthargie dans laquelle l'avait plongée Minsky, notamment. L'architecture de réseau de neurones dénommée « réseau de Hopfield » est devenue fort populaire, dans laquelle tous les neurones forment un graphe symétrique, étant tous connectés entre eux par des liens synaptiques symétriques. Sans doute, sa contribution la plus essentielle est d'avoir rapproché deux pôles scientifiques généralement tenus très éloignés : la physique des systèmes complexes et les sciences cognitives. Il a en effet démontré qu'un réseau de neurones connecté symétriquement converge naturellement vers ce que les physiciens dénomment un point fixe. Toute l'activité neuronale, après une phase transitoire dynamique, se stabilise et se fige dans un état donné.

Si on code dans cette activité neuronale un ensemble de « mémoires », par le biais d'un apprentissage de type « hebbien », dans lequel les liens synaptiques entre deux neurones se renforcent ou s'affaiblissent en fonction du rôle que ces neurones jouent dans les mémoires à coder (en substance, deux neurones verront leur lien se renforcer s'ils s'activent souvent ensemble ou, au contraire, s'affaiblir, si dans les mémoires à coder ils s'activent de manière antagoniste), ces mémoires deviendront les points fixes de la dynamique du réseau. Supposez, par exemple, comme dans la figure 21-2, que vous ayez codé les 10 premiers nombres entiers. Si, après l'apprentissage, vous rentrez comme valeur initiale des activités des neurones la même image mais quelque peu bruitée, le réseau convergera naturellement vers l'image apprise qui lui est la plus proche. Vous aurez ainsi réalisé une mémoire associative, capable de retrouver, au départ d'une image jamais rencontrée, celle qui lui correspond le plus dans l'ensemble des images apprises au départ. Que notre mémoire soit capable d'associer à un stimulus environnemental un concept mémorisé qui lui ressemble, voilà une donnée cognitive qu'il est difficile de refuser. Voilà également un type de programme idéal à concevoir dans un esprit et dans un langage OO. La contribution essentielle de Hopfield est de l'avoir incarné dans une théorie physique solide et mathématiquement analysable.

Plus récemment, Hopfield s'est détourné des points fixes comme mécanisme d'encodage de l'information pour s'intéresser à des formes plus dynamiques. En effet, les données neurophysiologiques tendent à prouver que notre cerveau passe plus de temps dans des dynamiques oscillatoires ou chaotiques que fixes. De plus, les neurones fonctionnement selon un mécanisme de type « intégration-déchargement » (dit également *spiking neuron* en anglais) où des pics d'intensité sont déchargés à une fréquence donnée, dépendant de l'intensité du stimulus reçu. La communauté neuronale pense de plus en plus que l'association cycle/chaos répond mieux à l'immense capacité d'encodage de notre cerveau, sa fabuleuse plasticité et sa faculté à retrouver très vite la réponse idéale à associer à une situation environnementale. Elle pense également que le cerveau est capable de traiter très finement les informations de nature temporelle véhiculées par les spiking neurons.

Tout d'abord : juste un ensemble d'objets

Une question : comment représente-t-on en programmation OO un ensemble d'objets ? La manière la plus immédiate consiste à stocker cet ensemble dans un tableau. Par exemple, supposons que nous cherchions à stocker dans un tableau un ensemble de neurones, sans aucunement se préoccuper, dans un premier temps, de la manière dont ils sont connectés. Nous commençons par définir une classe `Neurone`, caractérisée uniquement par son numéro et sa valeur d'activation.

```
public class Neurone {
  private int activation;
  private int numero;

  public Neurone(int numero) {
    activation = 0;
    this.numero = numero;
  }
  public void donneActivation() {
    System.out.println("l'activation du neurone " + numero + " est " +  activation);
  }
  public void changeActivation(int i) {
    this.activation = activation;
  }
}
```

On peut, n'importe où dans le code, créer un tableau de neurones de la manière suivante :

```
Neurone[] lesNeurones = new Neurone[100000] ;
```

Comme indiqué dans la figure qui suit, en Java ou C#, cette même écriture créera un premier objet dont le référent est `lesNeurones`, et qui contiendra l'adresse physique d'un objet possédant, lui-même, 100 000 adresses des neurones qu'il reste à créer.

Il faut noter, malgré tout, que ce tableau est typé par la classe des éléments qu'il contient. Il sera impossible par la suite de placer dans ce tableau autre chose que des neurones. Ensuite, il faut reproduire 100 000 fois l'instruction, pour chacun des neurones :

```
lesNeurones[i] = new Neurone(unNuméro) ;
```

Quand tous les objets sont installés dans un tableau, comme ici les neurones, il suffit pour accéder à l'un d'entre eux de connaître sa position dans ce tableau. Par exemple :

```
lesNeurones[5].donneActivation()
```

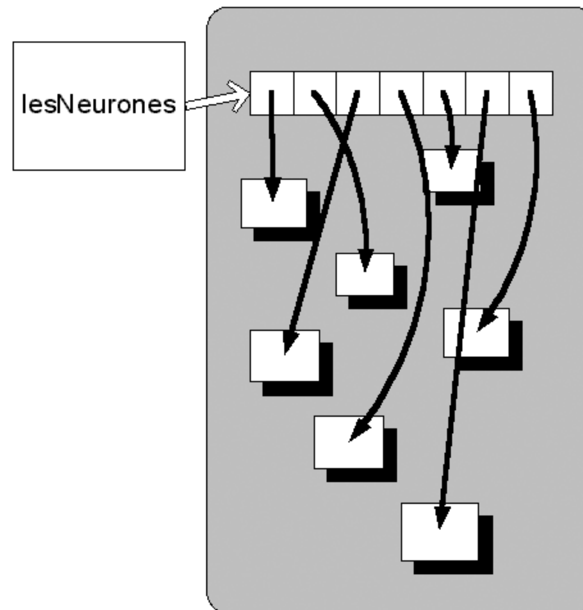


Figure 21-3
Stockage en mémoire d'un tableau de neurones.

Dans les langages de programmation permettant l'usage de la mémoire pile pour les objets, tous les objets compris dans un tableau seront regroupés dans la mémoire. Quand des mécanismes de mémorisation plus flexibles sont à l'œuvre, tel le stockage dans la mémoire tas, les objets peuvent être totalement dispersés. Toutefois, que ce soit dans la mémoire tas ou dans la mémoire pile, l'utilisation de tableaux pour stocker un ensemble d'objets peut nuire à la flexibilité et à l'économie. À la flexibilité, car il est nécessaire de définir dès le départ la taille du tableau.

Or, il est plus que fréquent, lors de l'écriture et l'exécution d'un programme, de vouloir se défaire de cette obligation, en permettant le rajout et la suppression d'objets en cours d'exécution, sans se limiter à une taille donnée. Si vous savez, au départ, le maximum de joueurs sur un terrain de foot, vous ignorez, en revanche, le nombre de buts, de cartons jaunes, de minutes de prolongation, et de dessous de table payés à l'arbitre. Notez que les neurones sont les seules cellules à ne plus pouvoir se reproduire à partir d'un certain âge, cette limite de taille les concerne donc un peu moins. On a vu en Java, comment l'utilisation de la classe `Vector` permet de lever cette limitation.

Quant au problème d'économie, il concerne surtout la mémoire pile, lorsque la suppression d'un petit tableau laisse un trou dans la mémoire difficile à combler par un autre tableau, de taille éventuellement plus grande. La mémoire se transforme graduellement en gruyère, maladie dont ont souffert beaucoup les disques durs il y a quelques années, eux qui sont sujets à des créations et effacements continus de fichiers de taille fort diverse. Défragmenter un disque dur, c'est, de fait, transformer ce gruyère en parmesan.

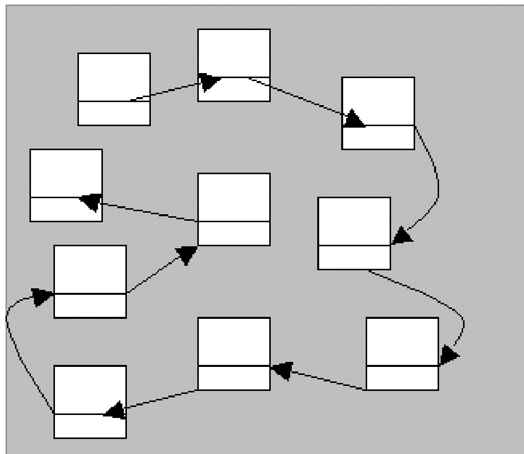
Liste liée

La parade que l'on a trouvée pour le stockage des fichiers dans le disque dur s'appelle une liste liée, et s'applique, en fait, pour n'importe quel stockage d'un groupe d'objets. Cela consiste à relier les objets installés, cette fois, n'importe où dans la mémoire, par une liste. Plus de limitation de taille et plus d'occupation massive et

compacte de la mémoire ne sont à craindre. Chaque objet de la liste contiendra, non seulement l'information le concernant, mais il rajoutera à ses attributs un référent pointant vers le prochain objet de la liste, comme indiqué dans la figure ci-après.

Figure 21-4

Une liste liée pour stocker un ensemble d'objets.



Cette liste liée sera évidemment d'autant plus utile qu'il existe, en effet, un ordre d'installation entre les objets, ordre que la succession des référents pourra facilement reproduire. En Java, la classe `Vector` (ou `ArrayList` en C# et Java) est réalisée par ce principe de liste liée, d'où son extensibilité. Une liste liée permet de s'affranchir de la taille maximale, et, de la manière la plus simple qui soit, d'ajouter ou de supprimer des éléments de la liste, au début, au milieu ou à la fin.

C'est également la meilleure parade contre les pertes de mémoires, vu qu'elle n'exige aucun regroupement en mémoire des objets à y installer. Dans le diagramme de classes suivant, apparaissent les trois classes nécessaires à la création de cette liste liée de neurones, ainsi que la manière dont ces trois classes se trouvent associées : la liste elle-même, les objets à stocker dans les éléments de la liste (ici, les neurones) et les éléments de la liste.

Chaque élément de la liste se caractérisera par un premier attribut associé à un neurone et par un second, établissant, par son référent, le lien avec l'élément suivant de la liste. Cela explique les liens d'association entre la classe `ElementNeuronal`, d'abord avec la classe `Neurone`, mais surtout avec elle-même, pour permettre la liaison entre chaque élément et son suivant dans la liste.

La liste nécessitera un premier attribut de type `ElementNeuronal` dont le rôle est de débuter cette liste, en tant que premier neurone de la liste. Un second attribut, le neurone courant, nous permettra de parcourir la liste. D'où les deux associations entre les classes `ListeNeuronale` et `ElémentNeuronal`. La liste contiendra également quelques méthodes (nous ne les détaillerons pas toutes ici, car elles sont communes à la programmation des listes liées), permettant d'éliciter les éléments de la liste, d'en installer un nouveau, au début, à la fin ou dans une position intermédiaire, d'en supprimer un, etc.

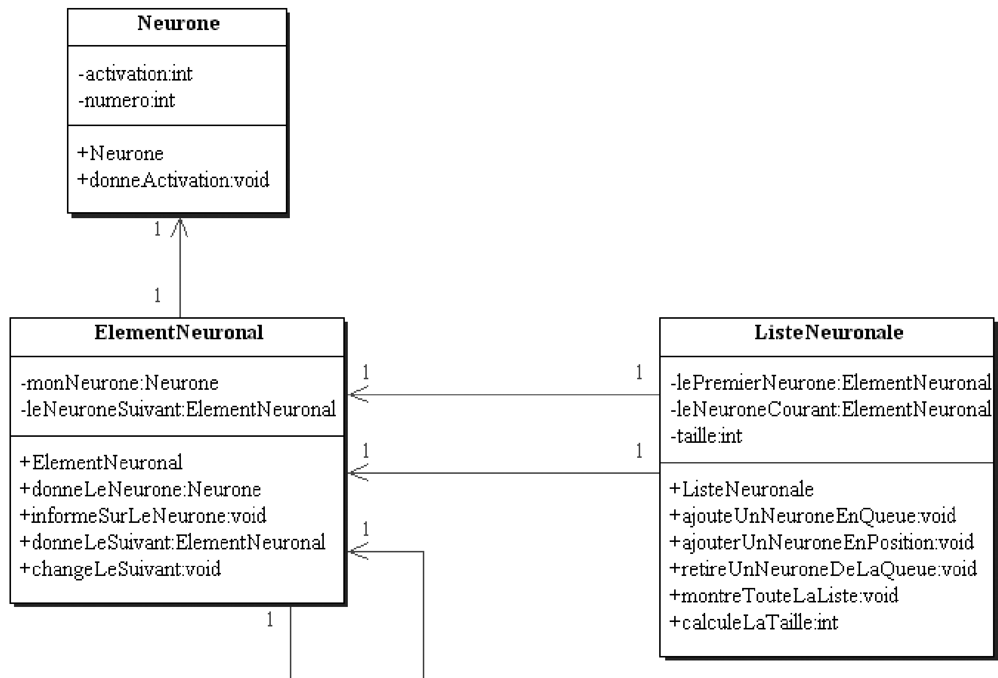


Figure 21-5
Diagramme de classe d'une liste liée.

Nous indiquons ci-après les codes Java et C++ ainsi que les résultats correspondant à ce diagramme. Le code C# se retrouve très facilement à partir du code Java. Ces deux codes créent une liste de quatre neurones d'activation nulle.

En Java

Fichier 1 : Neurone.java

```
public class Neurone {
  private int activation;
  private int numero;

  public Neurone(int numero) {
    activation = 0;
    this.numero = numero;
  }

  public void donneActivation() {
    System.out.println("l'activation du neurone " + numero + " est " + activation);
  }
  public void changeActivation(int activation)  {
    this.activation = activation;
  }
}
```

Fichier 2 : ElementNeuronal.java

```
public class ElementNeuronal {
  private Neurone monNeurone;
  private ElementNeuronal leNeuroneSuivant;

  public ElementNeuronal(Neurone monNeurone, ElementNeuronal leNeuroneSuivant) {
    this.monNeurone = monNeurone;
    this.leNeuroneSuivant = leNeuroneSuivant;
  }
  public Neurone donneLeNeurone () {
    return monNeurone;
  }
  public void informeSurLeNeurone() {
    monNeurone.donneActivation();
  }
  public ElementNeuronal donneLeSuivant() {
    return leNeuroneSuivant;
  }
  public void changeLeSuivant (ElementNeuronal leNeuroneSuivant) {
    this.leNeuroneSuivant = leNeuroneSuivant;
  }
}
```

Fichier 3 : ListeNeuronale.java

```
public class ListeNeuronale {
  private ElementNeuronal lePremierNeurone;
  private ElementNeuronal leNeuroneCourant;
  private int taille;

  public ListeNeuronale(Neurone lePremierDeLaListe) {
    lePremierNeurone = new ElementNeuronal(lePremierDeLaListe,null);
    this.leNeuroneCourant = lePremierNeurone;
    taille = 1;
  }
  public void ajouteUnNeuroneEnQueue(Neurone leNouveauNeurone)  {
    ElementNeuronal index = leNeuroneCourant;
    ElementNeuronal nouvelElement;

    while (index.donneLeSuivant() != null) {
      index = index.donneLeSuivant();
    }
    nouvelElement = new ElementNeuronal(leNouveauNeurone, null);
    index.changeLeSuivant(nouvelElement);
    leNeuroneCourant = nouvelElement;
    taille++;
  }
  public void ajouteUnNeuroneEnPosition (int i, Neurone leNouveauNeurone) {
    ElementNeuronal index = lePremierNeurone;
    int j = 1;
    while (++j < i) {
      index = index.donneLeSuivant();
```

```
    }
    ElementNeuronal unNouveau = new ElementNeuronal(leNouveauNeurone,
        index.donneLeSuivant());
    index.changeLeSuivant(unNouveau);
  }
  public void retireUnNeuroneDeLaQueue() {
    ElementNeuronal index = lePremierNeurone;

    while (index.donneLeSuivant().donneLeSuivant() != null) {
      index = index.donneLeSuivant();
    }
    index.changeLeSuivant(null);
    taille--;
  }
  public void montreTouteLaListe() {
    ElementNeuronal index = lePremierNeurone;
    while (index != null) {
      index.informeSurLeNeurone();
      index = index.donneLeSuivant();
    }
  }
  public int calculeLaTaille() {
    ElementNeuronal index = lePremierNeurone;
    taille = 0;

    while (index.donneLeSuivant() != null){
      index = index.donneLeSuivant();
      taille ++;
    }
    return taille;
  }
}
```

Fichier 4 : TestLaListeNeuronale.java

```
public class TestLaListeNeuronale {
 public static void main(String[] args) {
  Neurone n1 = new Neurone(1);
  Neurone n2 = new Neurone(2);
  Neurone n3 = new Neurone(3);
  Neurone n4 = new Neurone(4);

  ListeNeuronale uneListe = new ListeNeuronale(n1);
  uneListe.ajouteUnNeuroneEnQueue(n2);
  uneListe.ajouteUnNeuroneEnQueue(n3);
  uneListe.ajouteUnNeuroneEnPosition(2,n4);
  uneListe.montreTouteLaListe();
 }
```

Résultat

```
l'activation du neurone 1 est 0
l'activation du neurone 4 est 0
```

```
l'activation du neurone 2 est 0
l'activation du neurone 3 est 0
```

En C++

```
#include "stdafx.h"
#include "iostream.h"
class Neurone {
  private:
    int activation;
    int numero;
  public:
    Neurone(int numero){
      activation = 0;
      this->numero = numero;
    }
    void donneActivation(){
      cout << "l'activation du neurone " << numero << " est " <<
        activation << endl;
    }
    void changeActivation(int activation) {
      this->activation = activation;
    }
};
class ElementNeuronal {
  private:
    Neurone* monNeurone;
    ElementNeuronal* leNeuroneSuivant;
  public:
    ElementNeuronal(Neurone* monNeurone, ElementNeuronal* leNeuroneSuivant)
    {
      this->monNeurone = monNeurone;
      this->leNeuroneSuivant = leNeuroneSuivant;
    }
    Neurone* donneLeNeurone (){
      return monNeurone;
    }
    void informeSurLeNeurone(){
      monNeurone->donneActivation();
    }
    ElementNeuronal* donneLeSuivant(){
      return leNeuroneSuivant;
    }
    void changeLeSuivant (ElementNeuronal* leNeuroneSuivant){
      this->leNeuroneSuivant = leNeuroneSuivant;
    }
};
class ListeNeuronale{
  private:
    ElementNeuronal* lePremierNeurone;
```

```
    ElementNeuronal* leNeuroneCourant;
    int taille;
  public:
    ListeNeuronale(Neurone* lePremierDeLaListe){
      lePremierNeurone = new ElementNeuronal(lePremierDeLaListe, NULL);
      this->leNeuroneCourant = lePremierNeurone;
      taille = 1;
    }
    void ajouteUnNeuroneEnQueue(Neurone* leNouveauNeurone){
      ElementNeuronal* index = leNeuroneCourant;
      ElementNeuronal* nouvelElement;
      while (index -> donneLeSuivant() != NULL)
        index = index->donneLeSuivant();
      nouvelElement = new ElementNeuronal(leNouveauNeurone, NULL);
      index->changeLeSuivant(nouvelElement);
      leNeuroneCourant = nouvelElement;
      taille++;
    }
    void ajouteUnNeuroneEnPosition (int i, Neurone* leNouveauNeurone){
      ElementNeuronal* index = lePremierNeurone;
      int j = 1;
      while (++j < i)
        index = index->donneLeSuivant();
      ElementNeuronal* unNouveau = new ElementNeuronal(leNouveauNeurone, index->donneLeSuivant());
      index->changeLeSuivant(unNouveau);
    }
    void retireUnNeuroneDeLaQueue(){
      ElementNeuronal* index = lePremierNeurone;
      while (index->donneLeSuivant()->donneLeSuivant() != NULL)
        index ->donneLeSuivant()->donneLeSuivant();
      index->changeLeSuivant(NULL);
      taille--;
    }
    void montreTouteLaListe(){
      ElementNeuronal* index = lePremierNeurone;
      while (index != NULL) {
        index->informeSurLeNeurone();
        index = index->donneLeSuivant();
      }
    }
    int calculeLaTaille(){
      ElementNeuronal* index = lePremierNeurone;
      taille = 0;
      while (index->donneLeSuivant() != NULL) {
        index = index->donneLeSuivant();
        taille ++;
      }
      return taille;
    }
};
```

```
int main(int argc, char* argv[]) {
  Neurone n1(1);
  Neurone n2(2);
  Neurone n3(3);
  Neurone n4(4);
  ListeNeuronale uneListe(&n1);
  uneListe.ajouteUnNeuroneEnQueue(&n2);
  uneListe.ajouteUnNeuroneEnQueue(&n3);
  uneListe.ajouteUnNeuroneEnPosition(2,&n4);
  uneListe.montreTouteLaListe();
  return 0;
}
```

Résultats

```
l'activation du neurone 1 est 0
l'activation du neurone 4 est 0
l'activation du neurone 2 est 0
l'activation du neurone 3 est 0
```

La généricité en C++

Il est temps dans cet ouvrage de rendre justice à une des originalités du C++, un aspect syntaxique qui lui a longtemps été propre, et que son concepteur Bjarne Stroustrup, et pour cause, s'évertua à considérer comme un élément indispensable aux langages informatiques, et dans lequel il a vu pendant des années l'indéniable supériorité de sa créature sur tous les nouveaux venus. En effet, jusque récemment, tant Java que C# avaient préféré faire l'impasse sur ce mécanisme (la situation a depuis grandement évolué tant pour Java que pour C#, comme nous le verrons dans le prochain chapitre et donnant raison en cela à Stroustrup). La raison invoquée par les langages qui ont refusé de l'intégrer pendant des années était de préserver ainsi une facilité d'utilisation et de mise en œuvre que, paraît-il, l'utilisation de la « généricité » compromettait. Parmi les développeurs informatiques aussi, il n'y a que les C++ qui ne changent pas d'avis.

Dans l'opposition entre C++ et les autres, la généricité a longtemps été mise en avant afin de maintenir C++ sur la première place du podium. Il est donc important que vous compreniez, en quelques lignes, de quoi il retourne. Allons de l'avant avec la liste liée, pour introduire cette addition syntaxique, ce « supplément de généricité », également dénommé *template* en anglais et « modèle » en français.

On admettra volontiers que le mode de fonctionnement d'une liste est totalement indépendant de ce qui est contenu dans la liste. Une liste est un mode d'organisation, flexible et économique, nous l'avons vu, du stockage d'un ensemble d'objets, quel que soit le type d'objet concerné, des neurones, des voitures, des étudiants, des chiens et chats, enfin des objets quoi ! Toutes les fonctionnalités de la liste que nous avons spécifiées lors de la définition de la liste neuronale : `ajouteEnQueue`, `ajouteEnPosition(i)`, `retireDeLaQueue()`, `montreTouteLaListe`, `donneLaTaille()`, restent effectives quels que soient les objets qui y sont contenus.

De là, l'excellente idée du C++ de pouvoir définir ces fonctionnalités sans les limiter à un type d'objet précis. Observez attentivement le code C++ ci-après, et surtout comparez-le avec celui qui précède, et vous comprendrez la raison d'être de ce mécanisme original, ainsi que sa mise en œuvre.

Code C++

```
template <class T> /* T est un type générique, pas forcément une classe */
class ElementListe {
  private:
    T* monElement;
    ElementListe* leElementSuivant;
  public:
    ElementListe(T* monElement, ElementListe* leElementSuivant){
      this->monElement = monElement;
      this->leElementSuivant = leElementSuivant;
    }
    T* donneLeT (){
      return monElement;
    }
    void informeSurLeElement(){
      monElement->donneInformation();
    }
    ElementListe* donneLeSuivant(){
      return leElementSuivant;
    }
    void changeLeSuivant (ElementListe* leElementSuivant) {
      this->leElementSuivant = leElementSuivant;
    }
};
template <class T>
class ListeLiee {
  private:
    ElementListe<T>* lePremierElement;
    ElementListe<T>* leElementCourant;
    int taille;
  public:
    ListeLiee(T* lePremierDeLaListe){
      lePremierElement = new ElementListe<T>(lePremierDeLaListe, NULL);
      this->leElementCourant = lePremierElement;
      taille = 1;
    }
    void ajouteUnElementEnQueue(T* leNouveauElement){
      ElementListe<T>* index = leElementCourant;
      ElementListe<T>* nouvelElement;
      while (index -> donneLeSuivant() != NULL)
        index = index->donneLeSuivant();
      nouvelElement = new ElementListe<T>(leNouveauElement, NULL);
      index->changeLeSuivant(nouvelElement);
      leElementCourant = nouvelElement;
      taille++;
    }
    void ajouteUnElementEnPosition (int i, T* leNouveauElement) {
      ElementListe<T>* index = lePremierElement;
      int j = 1;
      while (++j < i)
        index = index->donneLeSuivant();
      ElementListe<T>* unNouveau = new ElementListe<T>(leNouveauElement,
          index->donneLeSuivant());
      index->changeLeSuivant(unNouveau);
    }
```

```
    void retireUnElementDeLaQueue(){
      ElementListe<T>* index = lePremierElement;
      while (index->donneLeSuivant()->donneLeSuivant() != NULL)
        index ->donneLeSuivant()->donneLeSuivant();
      index->changeLeSuivant(NULL);
      taille--;
    }
    void montreTouteLaListe() {
      ElementListe<T>* index = lePremierElement;
      while (index != NULL) {
        index->informeSurLeElement();
        index = index->donneLeSuivant();
      }
    }
    int calculeLaTaille() {
      ElementListe<T>* index = lePremierElement;
      taille = 0;
      while (index->donneLeSuivant() != NULL) {
        index = index->donneLeSuivant();
        taille ++;
      }
      return taille;
    }
};
int main(int argc, char* argv[]) {
  Neurone n1(1);
  Neurone n2(2);
  Neurone n3(3);
  Neurone n4(4);
  ListeLiee <Neurone> uneListe(&n1); /* on particularise la liste liée aux neurones */
  uneListe.ajouteUnElementEnQueue(&n2);
  uneListe.ajouteUnElementEnQueue(&n3);
  uneListe.ajouteUnElementEnPosition(2,&n4);
  uneListe.montreTouteLaListe();
  return 0;
}
```

Le résultat est le même que précédemment. Nous entamerons la description de ce code par le `main`, dans lequel nous créons une liste de neurones particulière, mais en la déclarant comme : `ListeLiee <Neurone>`, c'est-à-dire en exploitant la classe `Template ListeLiee`, tout en la particularisant pour des objets de type `Neurone`. En observant la description des deux classes à la base de la liste liée, la classe `ElementListe` et `ListeLiee`, nous constatons, de fait, que leur déclaration est précédée de `template <class T>`.

L'emploi du mot `class` est un peu malheureux ici, eu égard à l'utilisation de ce même mot pour la déclaration d'une classe, et vu également que ce mécanisme de « template » fonctionne quel que soit le type : une classe ou un type prédéfini (entier, réel…). Nous procéderions de la sorte pour une liste liée d'entiers ou de réels. En fait, cette instruction signifie simplement que `T` joue le rôle du type générique de l'objet, anciennement neuronal, mais que nous ne désirons plus préciser à ce stade-ci de la définition de la liste.

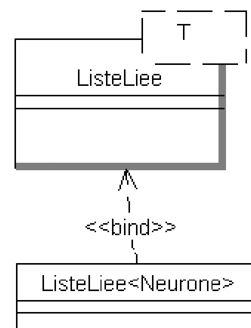
En comparant la version « générique » du code avec sa version « neuronale », vous constaterez que `T` a maintenant remplacé toutes les occurrences de `Neurone`. Et c'est bien ce qui permettrait de remplacer ce même `T` par `Voiture`, `Etudiant` ou `Chien`, pour autant que toutes ces classes aient été définies. En substance, pour passer de la classe `ListeNeuronale` à la classe `ListeLiee`, il nous a suffi de remplacer « neurone » par `T`. C'est comme en logique, lorsqu'on passe de la logique des propositions à celle des prédicats en introduisant les variables : on gagne un cran en généricité et en abstraction.

C'est lors de la compilation que le `T` disparaît, pour être remplacé par la classe qui désire bénéficier des services de la liste liée. Il s'agit donc essentiellement d'un jeu de récriture que le compilateur opérera en traduisant, pendant une phase préalable à la traduction en binaire, le code générique en un code particularisé pour les neurones. Bien évidemment, vu l'importance de la phase de compilation dans la mise en œuvre de la généricité, les deux langages de script Python et PHP 5 ont totalement fait l'impasse de ce mécanisme. Un chapitre entier serait nécessaire à la description détaillée de toutes les possibilités et de toute la richesse de cet emploi des « template » (on peut d'ailleurs également définir des fonctions « template »).

Il nous importe seulement que vous ayez compris l'esprit de la chose, cette possibilité de paramétrer des classes, de manière que leur comportement devienne indépendant de la nature des autres classes avec lesquelles elles pourraient interagir. Les classes dites « conteneurs » comme les listes liées, les vecteurs ou les graphes, sont en effet des candidats idéaux pour l'application de cette pratique. Il est possible en UML de représenter le lien qui unit la liste liée « template » à une instanciation particulière (ici neuronale).

Figure 21-6

En UML, la classe ListeLiee générique et sa particularisation neuronale.



La généricité en Java

Une telle généricité est-elle également possible en Java ou en C# ? Oui. Mais jusqu'à la version 1.4 de Java et dans les premières versions de C#, il fallait pour ce faire recourir à l'héritage et à la classe `Object`. En effet, cette classe est idéale pour prendre la place des objets auxquels la liste générique s'applique. La classe `Object` que nous avons détaillée au chapitre 14 sert à cela : intervenir quand on s'intéresse à des fonctionnalités universelles, communes à tous les objets.

En effet, en programmant le fonctionnement de la liste comme agissant sur la classe `Object`, il suffit, afin de récupérer ces mêmes fonctionnalités pour une classe plus spécifique, de procéder à quelques « casting ». Le code qui suit illustre l'utilisation de la classe `Vector` pour réaliser un « Vector de neurones ». « Vector » est un type de liste liée prédéfinie en Java, mais nous pourrions agir de même avec la classe `ListeLiee` que nous aurions programmée comme agissant sur la classe `Object`… Mais comme Java l'a fait pour nous, et que cela s'appelle un `Vector` ou `ArrayList`, pourquoi s'en priver ?

En Java

```
import java.util.*;
class VectorNeurone {
  private Vector unVecteur;
```

```
  public VectorNeurone() {
    unVecteur = new Vector();
  }
  public void ajouteUnNeuroneEnQueue(Neurone unNeurone) {
    unVecteur.addElement(unNeurone);
  }
  public void afficherLeNeuroneI(int i) {
    ((Neurone)unVecteur.elementAt(i)).donneActivation();
  }
  public void montreToutLeVecteur() {
    for (int i=0; i<unVecteur.size(); i++)
      ((Neurone)unVecteur.elementAt(i)).donneActivation();
  }
}
```

Quand on sait que Stroustrup est allergique à l'utilisation des « casting », obligatoire ici, car un « Vector » Java est avant tout un « Vector » d'objets de la classe `Object`, on comprend pourquoi il privilégiait sa pratique à celle de Java ou de C#. En clair, pour les amoureux des « typages forts », C++ offrait la meilleure solution, mais question lisibilité et compréhension du code, cela était discutable…

Depuis, la situation a nettement évolué car les versions les plus récentes de Java, depuis la version 1.5 tout autant que les versions les plus récentes de C#, depuis la deuxième, ont décidé d'inclure un mécanisme de généricité, totalement inspiré de celui du C++. Il permet par exemple de créer un Vector de neurones de la façon indiquée dans le petit code qui suit (pour C# vous pouvez procéder exactement de la même manière avec la classe `List` par exemple). On constate dans ce code que tous les « casting » ont disparu. C'est Stroustrup qui doit être content ! Mais attention, dorénavant le compilateur se fera un devoir de vérifier qu'il n'est plus possible de mettre autre chose que des « neurones » dans la liste, des neurones ou des sous-classes de ceux-ci. Finies les listes remplies de tout et n'importe quoi !

```
import java.util.*;

class Neurone {
 private int activation;
 private int numero;

 public Neurone(int numero) {
  activation = 0;
  this.numero = numero;
 }

 public void donneActivation() {
  System.out.println("l'activation du neurone " + numero + " est " + activation);
}
 public void changeActivation(int activation)  {
  this.activation = activation;
 }
}
```

```
public class TestGenericite {

         public static void main(String[] args) {
           Vector<Neurone> mesNeurones = new Vector<Neurone>(); /* attention, nouveau chez Java
           ➥depuis la  version 1.5 */
           mesNeurones.addElement(new Neurone(1));
           mesNeurones.elementAt(0).changeActivation(2); // plus de casting !
           mesNeurones.elementAt(0).donneActivation();
         }
}
```

Tant dans les nouvelles versions de Java que celles de C#, vous pouvez, tout comme dans le code C++ précédent, créer votre propre collection paramétrée. Ainsi, le code qui suit illustre en Java la création d'une liste liée paramétrée.

```
import java.util.*;

interface Element {
               void donneInformation();
}

class Neurone implements Element {
  private int activation;
  private int numero;

  public Neurone(int numero) {
    activation = 0;
    this.numero = numero;
  }

  public void donneInformation() {
    System.out.println("l'activation du neurone " + numero + " est " + activation);
  }
  public void changeActivation(int activation)  {
    this.activation = activation;
  }
}

class ElementListe <T extends Element> { // Attention vous devez hériter d'Element
    private T monElement;
    private ElementListe<T> leElementSuivant;

    public ElementListe(T monElement, ElementListe<T> leElementSuivant){
      this.monElement = monElement;
      this.leElementSuivant = leElementSuivant;
    }

    public T donneLeT (){
      return monElement;
    }
```

```
    public void informeSurLeElement(){
       monElement.donneInformation(); // pour pouvoir compiler cette instruction
    }

    public ElementListe<T> donneLeSuivant(){
      return leElementSuivant;
    }

    public void changeLeSuivant (ElementListe<T> leElementSuivant) {
      this.leElementSuivant = leElementSuivant;
    }
}

class ListeLiee <T extends Element> {
    private ElementListe<T> lePremierElement;
    private ElementListe<T> leElementCourant;
    private int taille;

    public ListeLiee(T lePremierDeLaListe){
      lePremierElement = new ElementListe<T>(lePremierDeLaListe, null);
      this.leElementCourant = lePremierElement;
      taille = 1;
    }

    public void ajouteUnElementEnQueue(T leNouveauElement){
      ElementListe<T> index = leElementCourant;
      ElementListe<T> nouvelElement;
      while (index.donneLeSuivant() != null)
        index = index.donneLeSuivant();
      nouvelElement = new ElementListe<T>(leNouveauElement, null);
      index.changeLeSuivant(nouvelElement);
      leElementCourant = nouvelElement;
      taille++;
    }

    public void ajouteUnElementEnPosition (int i, T leNouveauElement) {
      ElementListe<T> index = lePremierElement;
      int j = 1;
      while (++j < i)
        index = index.donneLeSuivant();
      ElementListe<T> unNouveau = new ElementListe<T>(leNouveauElement,
          index.donneLeSuivant());
      index.changeLeSuivant(unNouveau);
    }

    public void retireUnElementDeLaQueue(){
      ElementListe<T> index = lePremierElement;
      while (index.donneLeSuivant().donneLeSuivant() != null)
```

```
        index.donneLeSuivant().donneLeSuivant();
      index.changeLeSuivant(null);
      taille--;
    }

    public void montreTouteLaListe() {
      ElementListe<T> index = lePremierElement;
      while (index != null) {
        index.informeSurLeElement();
        index = index.donneLeSuivant();
      }
    }

    public int calculeLaTaille() {
      ElementListe<T> index = lePremierElement;
      taille = 0;
      while (index.donneLeSuivant() != null) {
        index = index.donneLeSuivant();
        taille ++;
      }
      return taille;
    }
}

public class TestGenericite {
            public static void main (String[] args) {
                          Neurone n1 = new Neurone(1);
                          Neurone n2 = new Neurone(2);
                          Neurone n3 = new Neurone(3);
                         Neurone n4 = new Neurone(4);
                          ListeLiee <Neurone>  uneListe = new ListeLiee <Neurone>(n1);
                         /* on particularise la liste liée aux neurones */
                          uneListe.ajouteUnElementEnQueue(n2);
                          uneListe.ajouteUnElementEnQueue(n3);
                          uneListe.ajouteUnElementEnPosition(2,n4);
                          uneListe.montreTouteLaListe();
            }
}
```

Une version C# serait en tout point semblable. Notez dans ce code la présence de l'interface `Element` ainsi que la manière dont le type paramétré hérite de cette interface de façon à pouvoir, sans entrave, déclencher le message `donneInformation` sur un objet de type `T`. Certaines facilités d'usage du C++ disparaissent du Java en raison de l'assignation dynamique qui caractérise son fonctionnement.

Il y aura lieu en Java comme en C# d'assister le compilateur en lui donnant quelques informations additionnelles sur le nature des classes qui viendront se substituer à l'exécution au paramètre T. Ainsi, dans le code précédent, le compilateur ne sait pas que c'est la classe `Neurone` qui viendra se substituer au paramètre T. Cette assignation se fait à l'éxécution.

Autre exemple, dans le code C# qui suit, il est nécessaire de prévenir le compilateur que la classe qui remplacera T contiendra en effet un constructeur sans argument (de manière à pouvoir effectuer l'instruction `new T()`

sans problème). Cela se fait dès la déclaration de la classe par le biais de la contrainte `where T : new()`. Cela n'est pas nécessaire en C++, car le compilateur aura créé la nouvelle classe en faisant les substitutions nécessaires et pourra le vérifier de lui-même. En Java comme en C#, la substitution finale s'effectue à l'exécution d'où des préoccupations additionnelles à prendre dès la compilation.

```
using System;
class MyClass {
  public MyClass() {
  }
}
class Test<T> where T : new() {
  T obj;

  public Test() {
    // Cela fonctionne grâce à la contrainte new().
    obj = new T(); // crée un objet T
  }
}

class ConsConstraintDemo {
  public static void Main() {
    Test<MyClass> x = new Test<MyClass>();
  }
}
```

Néanmoins, l'inspiration du C++ est plus qu'évidente. C'est à nouveau le compilateur qui procède à toutes les vérifications et effectue toutes les manipulations nécéssaires au bon fonctionnement de la généricité. Le code, une fois compilé, fonctionnerait avec les versions de Java et de C# antérieures à l'intégration de la généricité. Il paraît que cette compatibilité avec ses versions antérieures ne s'est pas faite sans mal en Java. Cela leur apprendra à vouloir faire simple quand on peut faire compliqué.

On comprendra aussi que ces deux langages ont prévu l'utilisation générique de certaines de leur collection comme les `ArrayList` dans leur librairie respective, tout comme nous le faisons ci-dessous pour la collection `ListeLiee` de notre invention. Cela justifie la possibilité d'un typage explicite pour la classe `Vector`, comme nous l'avons fait précédemment. Pour les raisons déjà discutées sur l'absence de typage statique, cette intégration devrait rester de l'ordre de l'impossible en Python et PHP 5.

Généricité

L'esprit de la généricité est partagé par tous les langages OO. Il s'agit d'autoriser le codage de fonctionnalités universelles, applicables sur tout type d'objet, et de pouvoir très facilement récupérer ces fonctionnalités pour un type donné cette fois. C++ l'a réalisé par un jeu d'écriture, qui permet au compilateur de particulariser un code générique à un type particulier. Lors de l'exécution, la fonctionnalité ne sera plus générique mais bien instanciée pour le type en question. En revanche, Java et C# recouraient jusqu'à présent à la classe `Object`, pour permettre le codage de cette fonctionnalité générique. Lors de l'exécution, il fallait s'assurer que le type particulier soit celui dans lequel les objets ont été « castés ». L'approche est plus intuitive et plus simple à mettre en œuvre, mais plus vulnérable en raison de la présence de « casting ». Ces deux langages proposent actuellement un mécanisme de généricité très proche de l'esprit du C++ qui résoud une fois pour toutes les problèmes de « casting » intempestif (ou, en son absence d'apparition de l'exception invalide `cast`) et permet d'éviter toutes les erreurs d'exécution qui peuvent en découler.

Passons aux graphes

La seule liste liée ne suffit cependant pas à coder un graphe ou un réseau de neurones, car, dans une liste, chaque neurone n'est connecté qu'à son seul voisin. Comme les neurologues se plaisent à dire que l'intelligence est fortement dépendante de la richesse des connexions neuronales, il nous importe d'augmenter le QI de notre petite structure. Chaque neurone devra être connecté, non plus à un seul neurone, mais à un groupe de neurones. Les schémas qui vont suivre peuvent, bien évidemment, s'appliquer à n'importe quel type d'objets, interconnectés entre eux par une structure de graphe.

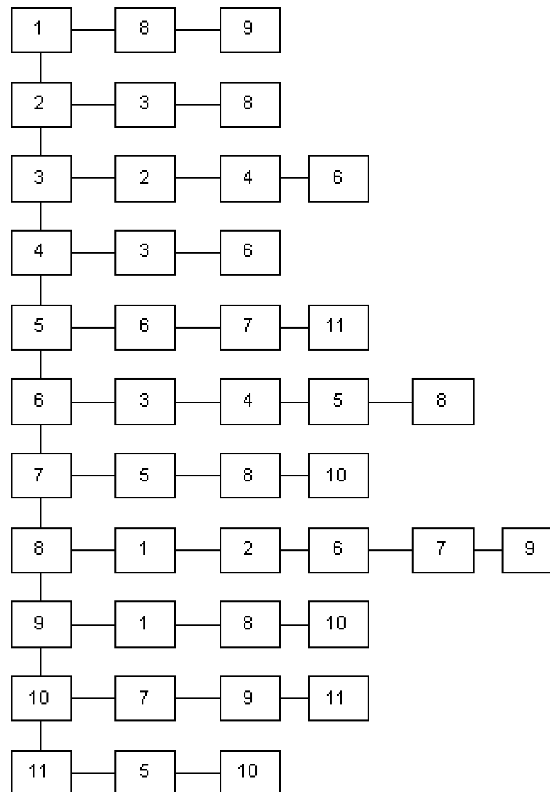


Figure 21-7
Réalisation d'une structure de graphe par l'utilisation croisée de deux listes liées.

Nous les limiterons dans la suite aux seuls neurones. Vous envisagez probablement une première solution possible, extension naturelle des listes liées vues précédemment, et représentée par la figure ci-après, pour le réseau illustré dans la figure au début du chapitre. Cette solution implique les classes indiquées dans le diagramme UML qui suit. Deux listes liées sont à considérer. La première, verticale dans la figure, permet d'éliciter tous les neurones, alors que la seconde, horizontale et associée à chaque nœud de la première, permet d'indiquer pour chacun des neurones à quels autres il est connecté (voir figures 21-7 et 21-8).

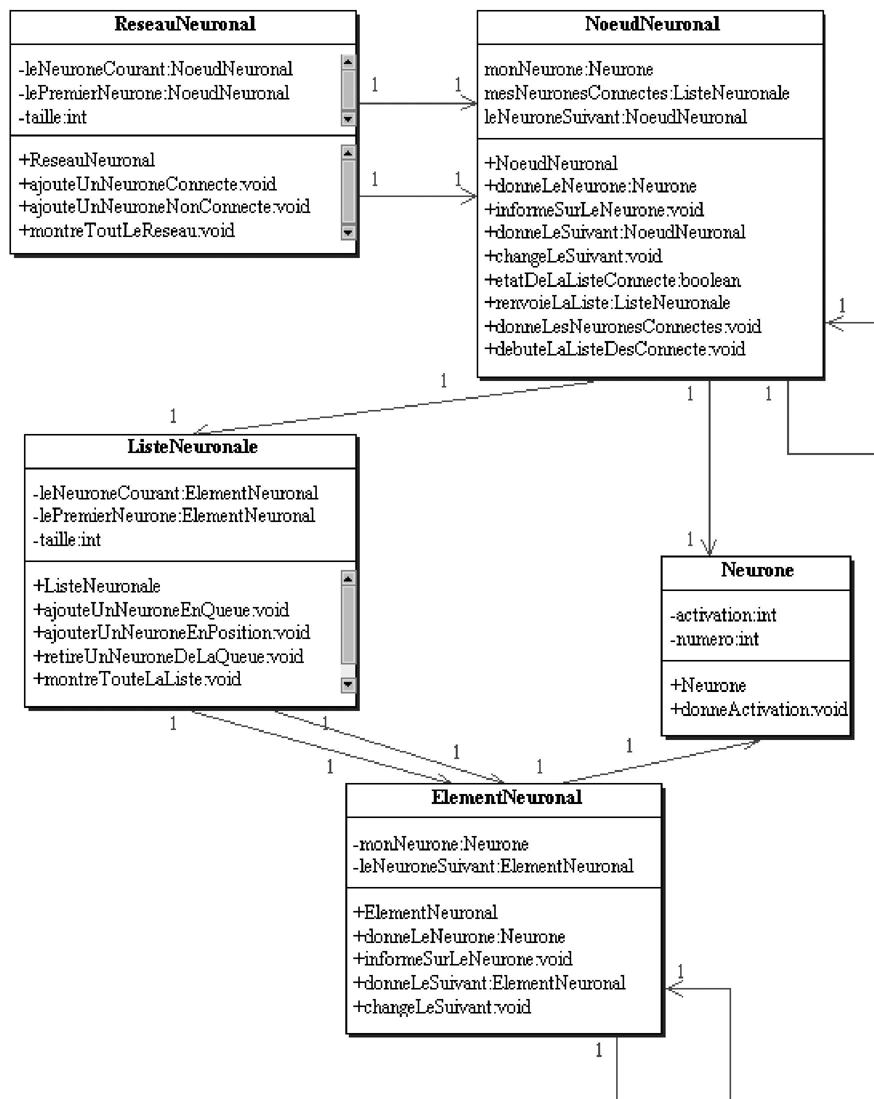


Figure 21-8
Diagramme de classe UML de la réalisation d'une structure de graphe par deux listes liées.

Dans le codage indiqué, chaque nœud du graphe (appelé ici nœud neuronal, qu'il faut distinguer d'élément neuronal qui s'applique aux éléments de la liste) possédera trois attributs, le neurone auquel il est associé, le nœud suivant (cela permet de coder la liste verticale de la figure 21-7), et la liste des neurones qui lui sont connectés (liste horizontale de la figure, et liste liée de neurones exactement semblable à celle traitée dans le chapitre précédent). Le code Java correspondant est donné ci-après. Nous n'indiquons que les deux classes additionnelles : `NœudNeuronal` et `ReseauNeuronal`.

Fichier NoeudNeuronal.java

```
public class NoeudNeuronal {
  private Neurone monNeurone;
  private ListeNeuronale mesNeuronesConnectes;
  private  NoeudNeuronal leNeuroneSuivant;

  public NoeudNeuronal(Neurone monNeurone, ListeNeuronale mesNeuronesConnectes, NoeudNeuronal
  ➥leNeuroneSuivant) {
    this.monNeurone = monNeurone;
    this.mesNeuronesConnectes = mesNeuronesConnectes;
    this.leNeuroneSuivant = leNeuroneSuivant;
  }
  public Neurone donneLeNeurone () {
    return monNeurone;
  }
  public void informeSurLeNeurone() {
    monNeurone.donneActivation();
  }
  public NoeudNeuronal donneLeSuivant() {
    return leNeuroneSuivant;
  }
  public void changeLeSuivant (NoeudNeuronal leNeuroneSuivant) {
    this.leNeuroneSuivant = leNeuroneSuivant;
  }
  public boolean etatDeLaListeConnecte() {
    if (mesNeuronesConnectes == null)
      return true;
    else
      return false;
  }
  public ListeNeuronale renvoieLaListe() {
    return mesNeuronesConnectes;
  }
  public void donneLesNeuronesConnectes() {
    if (mesNeuronesConnectes != null) {
      mesNeuronesConnectes.montreTouteLaListe();
    }
  }
  public void debuteLaListeDesConnecte (Neurone unNeurone) {
    mesNeuronesConnectes = new ListeNeuronale(unNeurone);
  }
}
```

Fichier ReseauNeuronal.java

```
public class ReseauNeuronal {
  private NoeudNeuronal leNeuroneCourant;
```

```
    private NoeudNeuronal lePremierNeurone;
    private int taille;

    public ReseauNeuronal (Neurone lePremierDuReseau) {
      lePremierNeurone = new NoeudNeuronal(lePremierDuReseau, null, null);
      this.leNeuroneCourant = lePremierNeurone;
      taille = 1;
    }
    public void ajouteUnNeuroneConnecte (Neurone leNeurone) {
      if (leNeuroneCourant.etatDeLaListeConnecte())
        leNeuroneCourant.debuteLaListeDesConnecte(leNeurone);
      else
        leNeuroneCourant.renvoieLaListe().ajouteUnNeuroneEnQueue(leNeurone);
    }
    public void ajouteUnNeuroneNonConnecte (Neurone leNeurone) {
      NoeudNeuronal unNouveauNoeud = new NoeudNeuronal(leNeurone,null,null);
      leNeuroneCourant.changeLeSuivant(unNouveauNoeud);
      leNeuroneCourant = unNouveauNoeud;
    }
    public void montreToutLeReseau() {
      NoeudNeuronal index = lePremierNeurone;
      while (index != null) {
        index.informeSurLeNeurone();
        index.donneLesNeuronesConnectes();
        index = index.donneLeSuivant();
      }
    }
}
```

Il serait possible également, pour coller parfaitement à la réalité des réseaux de neurones, de rajouter dans la classe `ElementNeuronal` un attribut « poids synaptique », reliant le `NoeudNeuronal` à l'`ElementNeuronal` de la liste. Bien que cette manière de coder un graphe par l'imbrication de deux listes soit très courante en informatique, nous lui préférons une autre manière, sans doute plus fidèle à la réalité que nous cherchons à reproduire. Ce nouveau codage, représenté par le dernier diagramme de classes, associe à chaque nœud neuronal une liste de neurones connectés.

Cependant, à leur tour, tous les éléments de la liste pourront être connectés à une nouvelle liste. Pour ce faire, chaque élément de cette liste se trouve être également un nœud neuronal. Les trois classes `NoeudNeuronal`, `ListeNeuronale` et `ElementNeuronal` forment une structure fermée, qui rend compte de la nature récursive de ce codage. Un nœud neuronal se connecte à une liste, mais elle-même, à son tour, est constituée d'éléments qui sont des nœuds neuronaux. La nature récursive des graphes paraît mieux prise en compte par ce nouveau codage. C'est celui que nous privilégions pour le codage des réseaux de neurones, et le même que nous retrouverons dans le chapitre suivant pour coder les molécules, comme autant de graphes d'atomes.

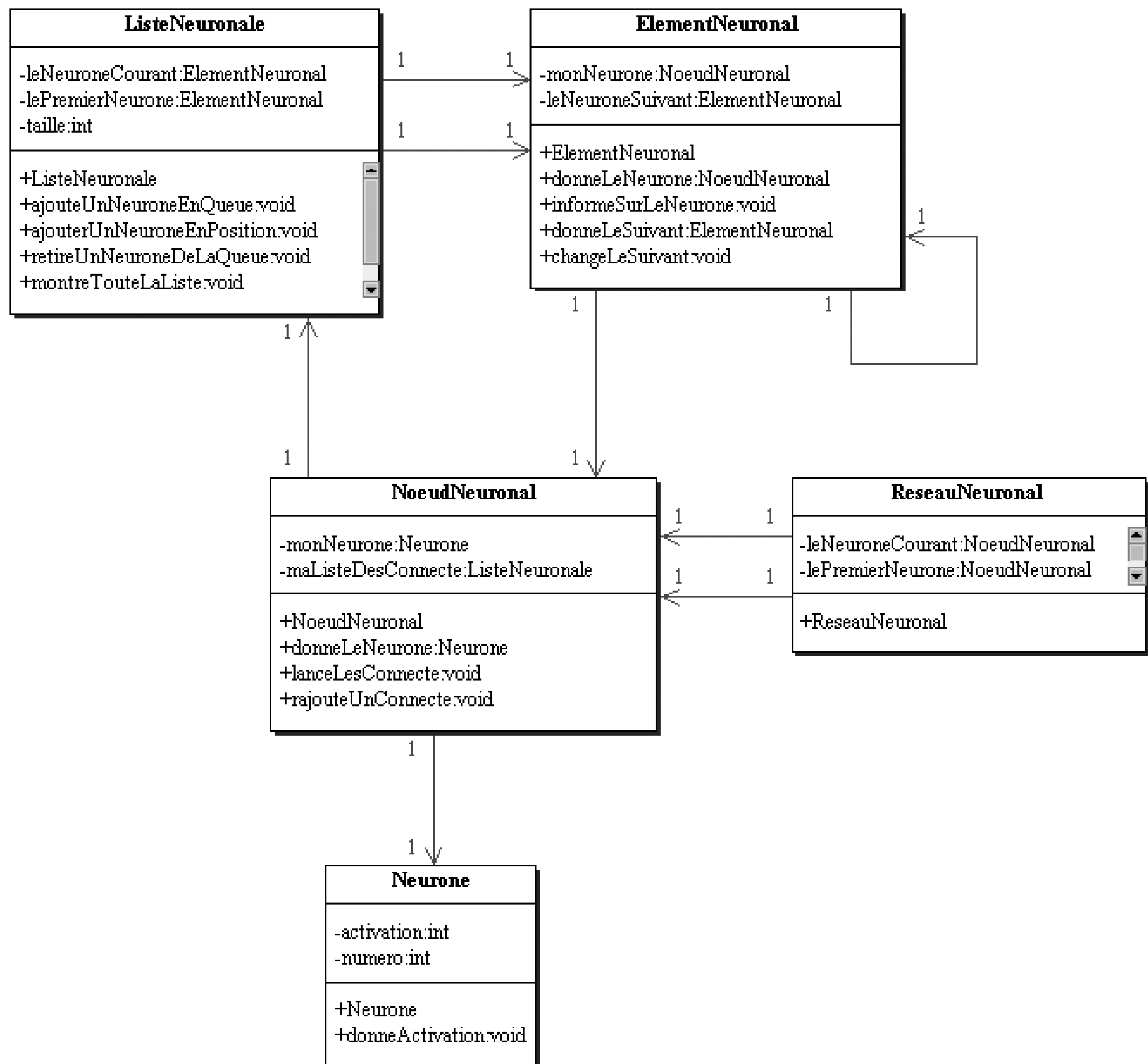


Figure 21-9

Diagramme de classe UML privilégié pour le codage d'un graphe.

Exercices

Exercice 21.1

Citez quatre exemples de graphe, deux directionnels et deux symétriques.

Exercice 21.2

Expliquez en quoi le principe de la liste liée favorise l'économie et l'adaptabilité pour le stockage d'un tableau d'objets.

Exercice 21.3

Réalisez en Java le code d'une liste liée d'objets « nombre entier » (utilisant la classe `Integer`).

Exercice 21.4

Expliquez pourquoi en C++ le codage d'une liste liée nécessite l'utilisation d'attribut de type pointeur.

Exercice 21.5

Expliquez pourquoi une possible solution pour le codage d'un graphe implique deux listes liées.

Exercice 21.6

Réalisez le diagramme de classes UML d'un réseau symétrique d'appareils électriques dans lequel ces appareils peuvent être de différent sous-type.

Exercice 21.7

Reprenez le dernier diagramme de classes du chapitre et insérez-y la classe `Synapse` qui relie deux neurones.

Exercice 21.8

Tentez de prédire ce qu'écrit à l'écran le programme C++ suivant :

```
#include "stdafx.h"
#include "iostream.h"
template <class T>
class Noeud {
  private:
    T valeurDuNoeud;
    Noeud<T> *unAutre;
    Noeud<T> *unAutreAutre;
  public:
    Noeud(T& valeur, Noeud<T> *unAutreNoeud=NULL, Noeud<T> *unAutreAutreNoeud=NULL):
      valeurDuNoeud(valeur),unAutre(unAutreNoeud), unAutreAutre(unAutreAutreNoeud)
    {}
```

```
    Noeud<T>* donneLautre() {
      return unAutre;
    }
    Noeud<T>* donneLautreAutre() {
      return unAutreAutre;
    }
    T donneLaValeur() {
      return valeurDuNoeud;
    }
    void metLaValeur(T& uneValeur) {
      valeurDuNoeud = uneValeur;
    }
    void attache(Noeud<T> *p){
      if (unAutre != NULL){
        p->unAutre = unAutre;
      }
      unAutre = p;
    }
    void attacheEncore(Noeud<T> *p) {
      if (unAutreAutre != NULL)
        p->unAutreAutre = unAutreAutre;
      unAutreAutre = p;
    }
    Noeud<T> *detache() {
      Noeud<T>* prochain = unAutre;
      if (prochain == NULL) return NULL;
      unAutre = prochain->unAutre;
      return prochain;
    }
    Noeud<T>* detacheEncore() {
      Noeud<T>* prochain = unAutreAutre;
      if (prochain == NULL) return NULL;
      unAutreAutre = prochain->unAutreAutre;
      return prochain;
    }
};
class Etudiant {
  private:
    char *nom;
    int cote;
  public:
    Etudiant(char* _nom, int _cote): nom(_nom),cote(_cote)
    {}
    void attribueLaCote(int _cote){
      cote = _cote;
    }
    int donneLaCote() {
      return cote;
    }
    char* donneLeNom() {
      return nom;
    }
};
```

```
int main() {
  Noeud<Etudiant> *etudiant1, *etudiant2, *etudiant3,
  *etudiant4, *etudiant5, *etudiant6,
  *etudiant7, *etudiant8, *etudiant9,
  *etudiant10;

  Etudiant *etudiantX = new Etudiant("Louis", 8);
  etudiant1 = new Noeud<Etudiant>(*etudiantX);
  etudiantX = new Etudiant("Louise",6);
  etudiant2 = new Noeud<Etudiant>(*etudiantX);
  etudiantX = new Etudiant("Juliette",7);
  etudiant3 = new Noeud<Etudiant>(*etudiantX);
  etudiantX = new Etudiant("Pascale", 6);
  etudiant4 = new Noeud<Etudiant>(*etudiantX, etudiant3, etudiant2);
  etudiantX = new Etudiant("Hugues", 5);
  etudiant5 = new Noeud<Etudiant>(*etudiantX, etudiant4);
  etudiantX = new Etudiant("Didier",2);
  etudiant6 = new Noeud<Etudiant>(*etudiantX);
  etudiant3->attache(etudiant6);
  etudiantX = new Etudiant("Jean",9);
  etudiant7 = new Noeud<Etudiant>(*etudiantX);
  etudiantX = new Etudiant("Jo",4);
  etudiant8 = new Noeud<Etudiant>(*etudiantX, etudiant7, etudiant3);
  etudiant6->attache(etudiant8);
  etudiantX = new Etudiant("Renaud", 2);
  etudiant9 = new Noeud<Etudiant>(*etudiantX, etudiant6);
  etudiantX = new Etudiant("Brigitte", 7);
  etudiant10 = new Noeud<Etudiant>(*etudiantX, etudiant5, etudiant9);
  etudiant7->attache(etudiant10);

  Noeud<Etudiant> *etudiantVoyage = etudiant4;

  for (int i=0; i<7; i++) {
    cout << etudiantVoyage->donneLaValeur().donneLaCote()<<"";
    etudiantVoyage = etudiantVoyage->donneLautre();
  }
  return 0;
}
```

Exercice 21.9

Tentez de comprendre la logique du petit programme C++ présenté ci-après et de prédire ce qu'il écrit à l'écran.

```
# #include "stdafx.h"
#include "iostream.h"
template <class T>
class Noeud {
  private:
    T valeurNoeud;
    Noeud<T> *suivant;
```

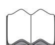
```
  public:
    Noeud(const T& item, Noeud<T> *noeudSuivant = NULL):
      valeurNoeud(item),suivant(noeudSuivant)
    {}
    Noeud<T> *getSuivant() {
      return suivant;
    }
    T getValeur() {
      return valeurNoeud;
    }
    void setValeur(const T& item) {
      valeurNoeud = item;
    }
    void lierA(Noeud<T> *p) {
      p->suivant = suivant;
      suivant = p;
    }
    Noeud<T> *delier() {
      Noeud<T> *noeudSuit = suivant;
      if (noeudsuit = NULL)
        return NULL;
      suivant = noeudsuit->suivant;
      return noeudsuit;
    }
    void ecritLaListe(Noeud<T> *debut) {
      Noeud<T> *courant;
      courant = debut;
      while (courant != NULL) {
        cout << courant->getValeur()<<endl;
        courant = courant->getSuivant();
      }
    }
};
int main() {
  Noeud<int> *p,*q,*r;
  p = new Noeud<int> (1, NULL);
  q = new Noeud<int> (2,p);
  r = new Noeud<int> (3,q);
  p->ecritLaListe(r);
  r->lierA(p);
  r->ecritLaListe(p);
  return 0;
}
```

22

Petites chimie et biologie OO amusantes

Tant la chimie que la biologie vont nous servir d'exemples pour la mise en pratique des principes de l'OO. Nous abordons ici ces deux disciplines de façon élémentaire, de manière à convaincre le lecteur de l'interêt de ces principes pour la réduction de la complexité inhérente à ces deux disciplines scientifiques. Deux petits simulateurs Java, le premier de réactions chimiques, le second de réactions immunologiques seront présentés.

Sommaire : Chimie computationnelle — Objets moléculaires — Objets réactionnels — Graphe moléculaire — Système immunitaire – Diagramme d'état-transition — Simulation



Doctus — À propos d'usine à gaz, je te propose de réfléchir à l'usage que tu pourrais faire d'objets simples dans un système reposant largement sur leurs interactions.

Candidus — Je te vois venir avec ton dada ! C'est vrai que, quand tu joues avec tes fioles et tes éprouvettes, tu ne manipules en fait que des éléments simples. Mais je n'ai jamais trouvé d'autre domaine que la chimie pour faire aussi compliqué avec si peu de choses !

Doc. — C'est bien ce que je te propose de remettre en question, justement. Comme tu viens de le dire, les éléments mis en jeu dans les processus chimiques peuvent, dans un premier temps, être modélisés très simplement.

Cand. — Et tu voudrais me faire la démonstration qu'avec l'OO les chimistes peuvent se passer de tubes à essai pour leurs expériences ?

Doc. — C'est le but ultime. Mais si les théories sont valables et les modèles expérimentaux suffisamment élaborés, l'OO est parfaitement prête à simuler toutes les réactions chimiques imaginables.

Cand. — Dis donc, mais c'est terrible ce que tu dis là ! On n'aura même plus besoin des ingrédients de départ d'une expérience pour obtenir les produits résultant d'une réaction chimique ! Quelle économie !

Doc. — Oui, si on veut. Si les ingrédients de départ sont virtuels, les produits finis le seront aussi !

Cand. — Pour les essais nucléaires, c'est pas un problème…

Doc. — Mais pour l'industrie pharmaceutique, c'en est un.

Cand. — Oh ! Y a bien le malade imaginaire…

Pourquoi de la chimie OO ?

Chimie computationnelle

Dans ce chapitre, nous allons présenter un travail qui occupe un des auteurs de cet ouvrage depuis quelques années, et qui se trouve au carrefour de deux disciplines : la chimie et la programmation orientée objet. Quelle drôle d’idée que celle de marier la chimie et la programmation ! À quoi cela sert-il de s’être spécialisé en programmation pendant ces dernières années ou de décider de s’y spécialiser, si c’est pour, dès l’achat d’un éventuel livre qui y soit consacré, se retrouver confronté à cette discipline exotique.

La chimie, pour laquelle beaucoup d’étudiants n’ont pas d’atomes crochus, se bornant à la considérer comme un passage obligé, semble éloignée de la programmation. Détrompez-vous, et ce à plus d’un titre. D’abord, parmi les multiples sous-disciplines qui constituent ce domaine des sciences naturelles, la chimie dite computationnelle est loin d’être en reste. La simulation informatique est aujourd’hui l’unique moyen de prédire la structure géométrique des molécules, ou encore le déroulement des réactions et l’évolution des concentrations moléculaires qui en découlent.

C’est également par la voie informatique que l’on peut coder ces molécules, les extirper d’une base de données, y compris à partir d’une information très incomplète ne reprenant qu’une partie de leur structure, étudier de nouvelles configurations moléculaires et tenter de prédire leur stabilité et leur propriété (toute la pharmaceutique est ici concernée).

Chimie comme plate-forme didactique

Mais si nous nous intéressons ici à la chimie, dans un livre dédié aux principes de programmation OO, c’est que cette discipline nous semble fournir une plate-forme didactique idéale pour la présentation des mécanismes OO. À l’instar de la biologie, qui regorge d’objets en interaction, comme nous avons déjà pu l’apprécier tout au long de cet ouvrage (rappelez-vous ce que doit Alan Kay, un des précurseurs de l’OO, à la biologie), la chimie nous offre plusieurs catégories d’objets intéressantes, dont l’établissement des relations et la résolution de l’organisation taxonomique sont autant de précieux et très didactiques exemples. De plus, la chimie manque aujourd’hui de modélisations informatiques formalisées, qui permettraient d’assister les chimistes dans les nombreux problèmes de nomenclature et de classification qu’ils rencontrent, tant des espèces chimiques que des schémas réactionnels. L’OO, notamment par l’utilisation des diagrammes UML, permet à un chimiste, pas forcément versé dans les joies divines de la programmation, de participer à l’élaboration des modèles informatiques.

Nous avons des durillons aux doigts à force de l’avoir écrit, mais l’OO constitue à ce point un progrès en réconciliant programmation et sens commun qu’il serait égoïste de ne pas en faire profiter d’autres disciplines telles que la chimie.

Une aide à la modélisation chimique

Enfin, comme une modélisation informatique n’arrive jamais seule, mais mène naturellement à l’exécution d’un programme de simulation, il n’est pas impossible que des tentatives de modélisation, telles que celle introduite ici, puissent contribuer (en complément à tout ce que la chimie computationnelle fournit déjà comme outil) à aider les chimistes, soit à une prédiction plus fine de certaines situations chimiques, ou, plus simplement, à la mise en évidence et l’illustration expérimentale de certains principes qualitatifs bien connus d’eux. Cette dernière raison d’être du simulateur que nous allons présenter ne va pas sans rappeler les travaux de simulation de Walter Fontana, simulation dénommée « Alchemy » pour « Algorithmic Chemistry ».

C'est un réacteur chimique que le simulateur OO présenté ici cherchera à reproduire. À partir de quelques molécules et schémas réactionnels initiaux, le simulateur permettra de suivre l'apparition de nouvelles molécules, ainsi que l'évolution dans le temps de leur concentration. Ce livre reste avant tout un livre de programmation, et n'exigera de votre part, pour comprendre ce qui suit, qu'une connaissance élémentaire de chimie, le genre de connaissance que l'on acquiert pendant les premières années de lycée mais que, généralement, sauf à prolonger des études scientifiques, on s'empresse d'oublier.

Certains détails seront volontairement laissés sous silence, dès le moment où ils demandent quelques connaissances chimiques plus poussées, ou qu'ils nécessitent un niveau d'analyse qui dépasse la mission d'un livre d'initiation à l'OO. Il faudra sans doute, çà et là, faire un petit effort de mémoire, mais toute votre réflexion devra porter sur la mise en place de la structure OO, et en rien sur les concepts chimiques abordés.

Walter Fontana, Irun Cohen

Walter Fontana a très longtemps été chercheur au célèbre Institut Santa Fe, que nous avons déjà évoqué dans l'encart dédié à Stuart Kauffman, et travaille désormais à Harvard. De fait, les travaux de ce chercheur ne vont pas sans rappeler ceux de Kauffman, avec en ligne de mire une meilleure compréhension de l'émergence du vivant et de la nature intime des systèmes complexes. Fontana est l'auteur d'Alchemy (Algorithmic Chemistry), un programme qui fut une source d'inspiration essentielle aux travaux présentés dans ce chapitre. Dans Alchemy, un ensemble d'objets chimiques rentre dans des réactions afin de produire de nouveaux objets. Ce travail est à l'origine d'un courant de recherche récent dénommé « Chimie artificielle », et dans lequel on retrouve, comme dans ce chapitre, deux types d'objets essentiels : les composants chimiques et les réactions dans lesquelles ils entrent.

Le but de ce type de simulation est de donner naissance à des structures complexes, telles que des molécules ou des réseaux réactionnels, capables et de s'auto-maintenir et d'évoluer. Par exemple, un réseau réactionnel s'auto-maintient, si toutes les molécules participant à ce réseau sont à la fois réactantes et produites par ce même réseau. Ce réseau sera par essence stable, car tout retrait d'un des éléments du réseau ne peut conduire qu'à sa régénération par ce même réseau. De même, au départ de quelques éléments du réseau, sa complète reconstruction est plus que probable.

Selon Fontana, l'auto-maintien est une condition première à l'émergence du vivant. Tant la reproduction que l'évolution ne peuvent se produire si cet auto-maintien n'est pas au départ garanti. Il faut, en effet, qu'une structure soit capable d'une certaine stabilité pour pouvoir évoluer vers de nouvelles structures, tout aussi stables mais gagnant en complexité.

Irun Cohen est un formidable immunologiste de l'Institut Weismann à Tel Aviv qui, conscient de l'immense et irréductible complexité du système qu'il étudie, effectue un véritable travail de croisade. Assisté de l'inventeur des diagrammes d'état-transition David Harel, il entend convaincre les immunologistes de recourir aux outils orientés objet. Il est également un des chercheurs les plus spécialisés dans la compréhension et le traitement des maladies auto-immunes, maladies qui dépassent la plupart des immunologistes par leur apparente étrangeté et grande complexité.

Les diagrammes de classe du réacteur chimique

La présentation que nous allons faire de notre petite chimie OO se limitera essentiellement à l'exposé et la discussion des diagrammes de classe UML. Cependant, toute la simulation a été réalisée en Java, et nous présenterons à la fin du chapitre certains résultats obtenus en l'exécutant, pour illustrer les composants essentiels de notre programme et le fonctionnement du simulateur. Nous allons présenter, dans un premier temps, et l'une après l'autre, les classes principales constituant le simulateur. Nous montrerons également les diagrammes de classe correspondants, et ce de manière progressive. Mais avant tout, il sera important de séparer deux ensembles de classes distinctes, que nous pourrions placer dans deux « assemblages » distincts : d'une part, les composants chimiques et, d'autre part, les réactions chimiques.

Les composants chimiques

La classe composant chimique

La première classe reprend tous les composants chimiques de notre simulateur.

Figure 22-1
La classe composant chimique.

Composant Chimique
-symbole:String
-identite:int
-manque:int
-concentration:int
-concentrationChange:int
+metAJourConcentration:void
+equals:void
+decroitLeManque:void
+accroitLeManque:void
+suisJeRadical:void
+ecritLeSymbole:void

Découvrons-en les attributs :

- `symbole` : cet attribut reprend le symbole ou la formule chimique du composant. Il s'agit d'un attribut de type `String`. De façon classique, la chimie décrit ces composants comme « O_2 », « H_2O », « Na+ », etc., mais nous verrons dans la suite la manière plus simple et « moins chimique », que nous avons adoptée pour coder ces composants.
- `identite` : il s'agit de pouvoir identifier le composant par un index entier unique. Cet entier pourrait, dans le cas des atomes, être relié à ce qui caractérise l'atome de manière unique, comme son numéro atomique. Cet attribut joue le même rôle que la clé primaire dans une base de données. Il nous permettra également d'ordonner les atomes entre eux, lors des problèmes de canonisation que nous discuterons dans la suite.
- `manque` : cet attribut indique le nombre d'électrons dépareillés dans le composant. Dans toute molécule, les électrons se doivent d'être en couple et, quand cela n'est pas le cas, la molécule se débrouillera pour combler ce manque. En général, cet attribut ne peut prendre que trois valeurs : 0, 1 (on parle alors d'un composant radical, comme l'hydrogène) et 2 (on parle alors d'un bi-radical, comme l'oxygène). Cet attribut est important car, lors d'une réaction chimique, il se doit d'être conservé dans le passage des réactants aux produits.
- `concentration` : nous nous intéressons à la simulation de réactions chimiques, pendant lesquelles la concentration des composants chimiques varie sans cesse. La présence de cet attribut vous aide à bien comprendre que chaque objet de la classe `Composant Chimique` représente un composant donné, tel l'oxygène ou le méthane, la concentration servant à quantifier son nombre. Cette concentration varie lors de chaque réaction dans laquelle rentre ce composant. Il ne s'agit pas d'avoir, par exemple, une classe `Oxygene` ou `Methane`, et un objet par molécule d'oxygène ou de méthane. Nous ne voulons pas saturer la mémoire de l'ordinateur avec une multiplication effrénée d'objets parfaitement identiques, sauf peut-être leur distribution dans l'espace, mais que de toute façon nous ne considérons pas ici.
- `concentrationChange` : à chaque pas de temps de la simulation, cet attribut reprend tous les changements subis par cette concentration. C'est cette valeur qui servira à mettre à jour la valeur de l'attribut `concentration`.

Parmi les méthodes les plus importantes, certaines s'expliquent d'elles-mêmes, comme `miseAJourConcentration()`. `ecritLeSymbole()` est une méthode abstraite qui découvre et écrit le symbole du composant, à partir de sa composition et de sa structure. Les autres méthodes sont concrètes. Une première permet de comparer deux composants entre eux à partir de leur symbole, elle renvoie « true » si les symboles sont égaux. Cette méthode est, en fait, la redéfinition de la méthode `equals`, héritée de la super-superclasse `Object`, en adoptant une pratique décrite dans le chapitre 14 dédié à cette classe.

Deux méthodes s'occupent de modifier l'attribut `manque` du composant. Cette modification se produit en réalité si le composant capture ou perd un électron. Remarquons que, grâce à l'encapsulation des attributs, nous préserverons l'intégrité des molécules, en nous assurant que ces deux méthodes agissent de telle sorte que la valeur de l'attribut `manque` soit toujours 0, 1 ou 2.

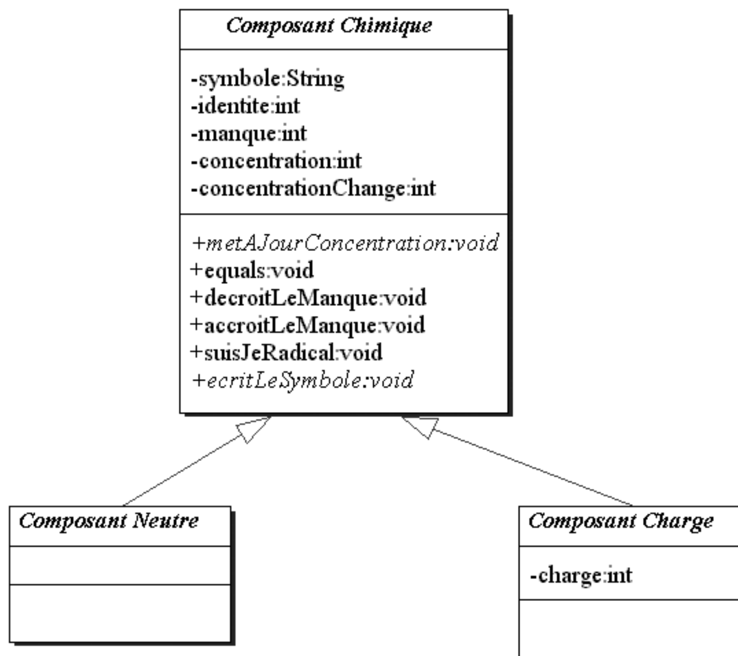
Les classes composant neutre et composant chargé

À partir de cette première superclasse abstraite (eh oui ! il n'existe pas non plus dans la nature de composants chimiques), deux nouvelles sous-classes héritent, toujours abstraites, nous permettant de séparer les composants chargés (ou ioniques) des composants neutres. Pour l'instant, seule l'addition de l'attribut `charge`, un entier positif ou négatif, distingue les deux sous-classes. Rien ne nous empêcherait, pour simplifier la conception, de ne rester qu'avec notre superclasse, pour laquelle l'addition de ce seul attribut permettrait de séparer les objets composants neutres (présentant une valeur nulle pour cet attribut) des objets composants chargés (présentant une valeur positive ou négative pour cet attribut).

Néanmoins, il est probable que l'existence de cette charge rende le comportement des composants chargés à ce point différent des composants neutres (par exemple, si vous les placez dans une électrode ou dans un champ électrique) que cette distinction reste utile. On se trouve devant un problème de conception qu'il est difficile à ce stade de résoudre. Y a-t-il lieu de créer ces deux sous-classes ? Seul un vrai chimiste pourrait répondre à la question.

Figure 22-2

Ajout des deux sous-classes Composant_Neutre et Composant_Charge.



C'est là que l'on comprend tout l'intérêt de la programmation OO pour la chimie. Grâce à la vision formelle de l'héritage proposée par l'OO, le chimiste verra s'il est utile ou pas de considérer un composant ionique comme une sous-classe des composants chimiques. En tous les cas, pour nous, la question reste ouverte. La rigueur de l'OO forcera le chimiste à clarifier ses taxonomies chimiques.

La classe `Composant_Neutre` et la classe `Composant_Charge` vont donner lieu, chacune, à trois sous-classes. D'abord, décrivons les trois sous-classes de la première : `Atome`, `Molecule` et `Groupe`.

La classe atome

`Atome` est une sous-classe de composant neutre. Notre première sous-classe concrète. Soyez tout à fait rassurés, il existe bien des atomes dans la nature, on ne vous a pas menti. Comme nouvel attribut additionnel, nous trouvons sa valence. Chaque atome présente une tendance naturelle à remplir ou à vider sa couche électronique finale d'un certain nombre d'électrons, afin de rejoindre l'élément stable qui lui est le plus proche dans le tableau de Mendeleïev.

L'hydrogène a une valence 1, signifiant qu'il a tendance à récupérer un électron, l'oxygène 2 (d'où H_2O), le carbone 4 (d'où le méthane CH_4). Comme méthode, nous concrétisons d'abord la méthode `ecritLeSymbole`, qui écrira le symbole associé à l'atome, et qui se limite à être un nombre entier comme nous allons le voir. Une autre méthode est héritée de l'interface `cloneable`, il s'agit de `clone`, que nous redéfinissons dans `Atome` et qui permettra de dupliquer ce dernier.

L'identité de tout objet `atome` sera un nombre entier propre à chaque atome. Dans notre simulateur, pour des raisons liées à la structure computationnelle des molécules, la valeur de cette identité sera liée au numéro atomique et à la valence de l'atome. Plus cette valence sera petite, plus grande sera l'identité. Jusqu'ici, nous n'avons considéré que 5 objets `atomes`, mais, bien sûr, nous pouvons en traiter autant qu'on veut.

Ce sont un atome d'identité 1 et de valence 4 (que nous pourrions assimiler au carbone), un atome d'identité 2 et de valence 3 (que nous pourrions assimiler à l'azote), un atome d'identité 3 et de valence 2 (que nous pourrions assimiler à l'oxygène), un atome d'identité 4 et de valence 1 (que nous pourrions assimiler à l'hydrogène) et un autre d'identité 5 mais également de valence 1 (et que nous pourrions assimiler au chlore). Pour l'atome, la valeur de l'attribut `symbole` se borne à reproduire cette même identité.

La classe molécule

Parmi les attributs de la classe `Molecule`, nous trouvons deux tableaux, le premier d'atomes, reprenant tous les atomes présents dans la molécule, et le second d'entiers et de la même dimension, reprenant, pour chaque atome, le nombre de fois qu'il apparaît dans la molécule. Nous aurions pu stocker cette information de bien d'autres manières, mais nous restons délibérément le plus simple qui soit.

D'autres attributs référents présents dans la molécule résulteront des associations que la molécule possède avec deux autres classes, qu'ils nous restent à définir : la classe `Liaison` et la classe `NoeudAtomique`. Nous les définirons et les installerons plus tard dans le diagramme UML. La molécule stockera deux tableaux de liaisons, celles qui se brisent le plus facilement, et celles qui s'ouvrent le plus facilement. De manière à coder la molécule comme un graphe computationnel, pareil à ceux décrits dans le chapitre précédent, il suffira qu'elle soit associée au premier nœud atomique qui la constitue.

Parmi les méthodes de la classe `Molecule`, nous trouvons un nombre très important de constructeurs. En effet, dans notre simulateur, une molécule peut apparaître de multiples façons, en fonction des réactions qui lui donnent naissance. Pour pratiquement chacune des réactions, nous devrons définir un constructeur de la molécule. Nous savons que ce qui différencie plusieurs constructeurs est le nombre ou la nature des arguments. Un exemple suffira à en justifier la multiplicité.

Une molécule peut naître d'une réaction brisant une molécule existante ou de la combinaison de deux autres molécules. Pour simplifier, dans le premier cas, le constructeur de la nouvelle molécule recevra comme argument

la molécule de départ, alors que, dans le second cas, ce constructeur recevra les deux molécules de départ. Deux méthodes permettent de retrouver les liens les plus faciles à briser et à ouvrir. Deux méthodes permettent d'augmenter ou de diminuer la radicalité de la molécule et portent, en conséquence, sur l'attribut `manque`.

La classe Groupe

Un groupe atomique n'est rien d'autre qu'un assemblage d'atomes, très proche d'une molécule. Cependant, il n'est jamais une molécule en soit, mais est toujours partie intégrante d'une molécule. Il sert surtout à la caractérisation des molécules et joue un rôle important dans la définition et le type des réactions. Pour toutes ces raisons, nous n'irons pas plus loin dans la description des groupes.

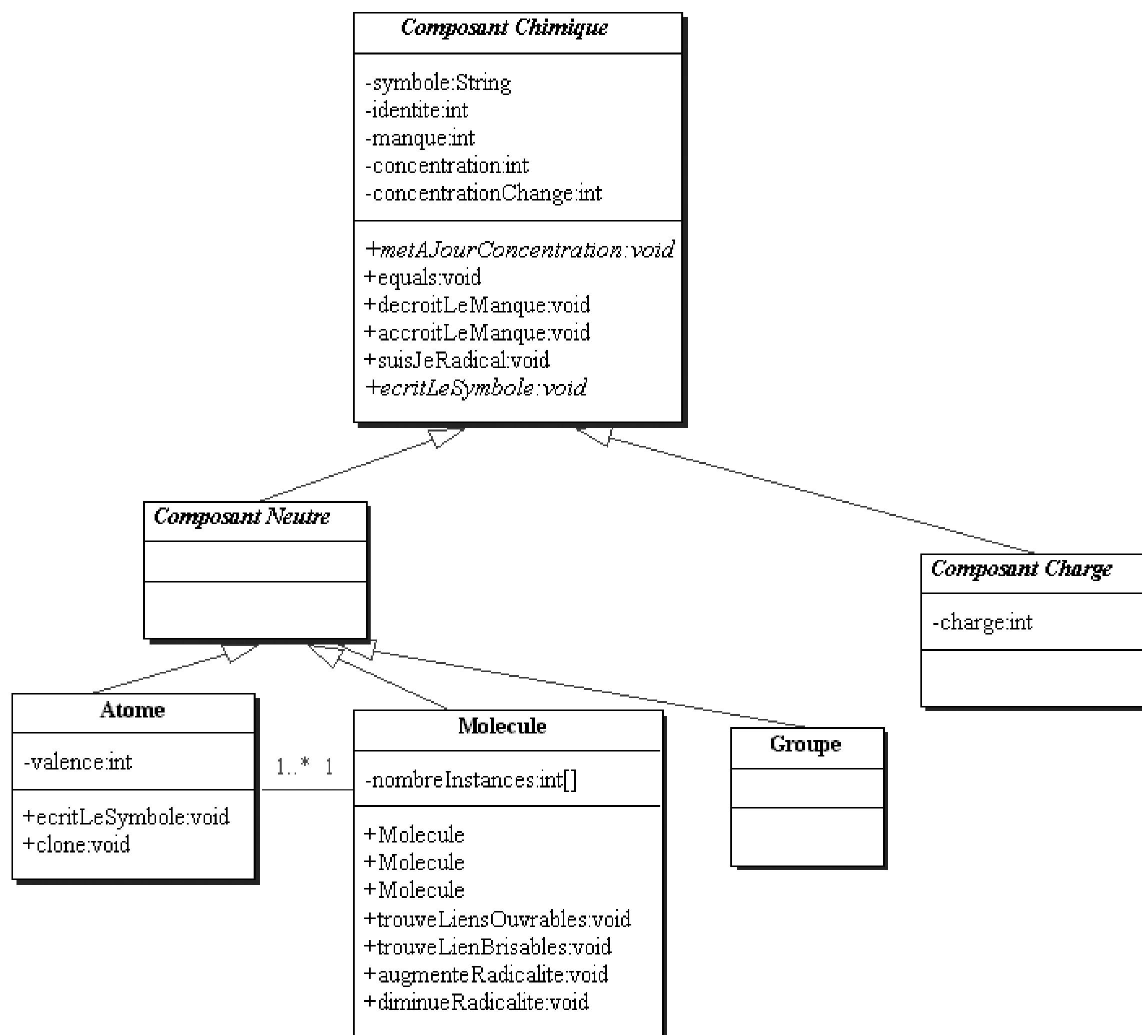


Figure 22-3

Ajout dans le diagramme UML des trois sous-classes de Composant_Neutre, les classes Atome, Molecule et Groupe.

Les trois sous-classes de composant chargé

Il nous faut maintenant définir et particulariser les trois sous-classes de `Composant Charge`, la version chargée des trois sous-classes que nous venons de définir : `IonAtomique`, `IonMoleculaire`, `GroupeIonique`. Dans un langage autre que Java, et permettant le multihéritage, comme C++, on pourrait décemment se demander pourquoi ne pas faire hériter chacune de ces classes de deux sous-classes : la classe `Composant Charge` et la classe neutre correspondante.

Par exemple, la classe `IonAtomique` hériterait de la classe `Composant Charge` et de la classe `Atome`. Si ce n'est pas en soi une mauvaise idée, il faudrait être toutefois très attentif à définir cet héritage comme « virtuel », car nous nous retrouverions exactement dans une situation dont nous savons, depuis le chapitre 11, qu'elle pose problème. Nous retrouverions deux fois, dans le graphe d'héritage, la classe `Composant Chimique`, une fois comme superclasse de `Atome` et une fois comme superclasse de `Composant Charge`. Or, il ne doit se retrouver qu'une seule fois dans chaque atome.

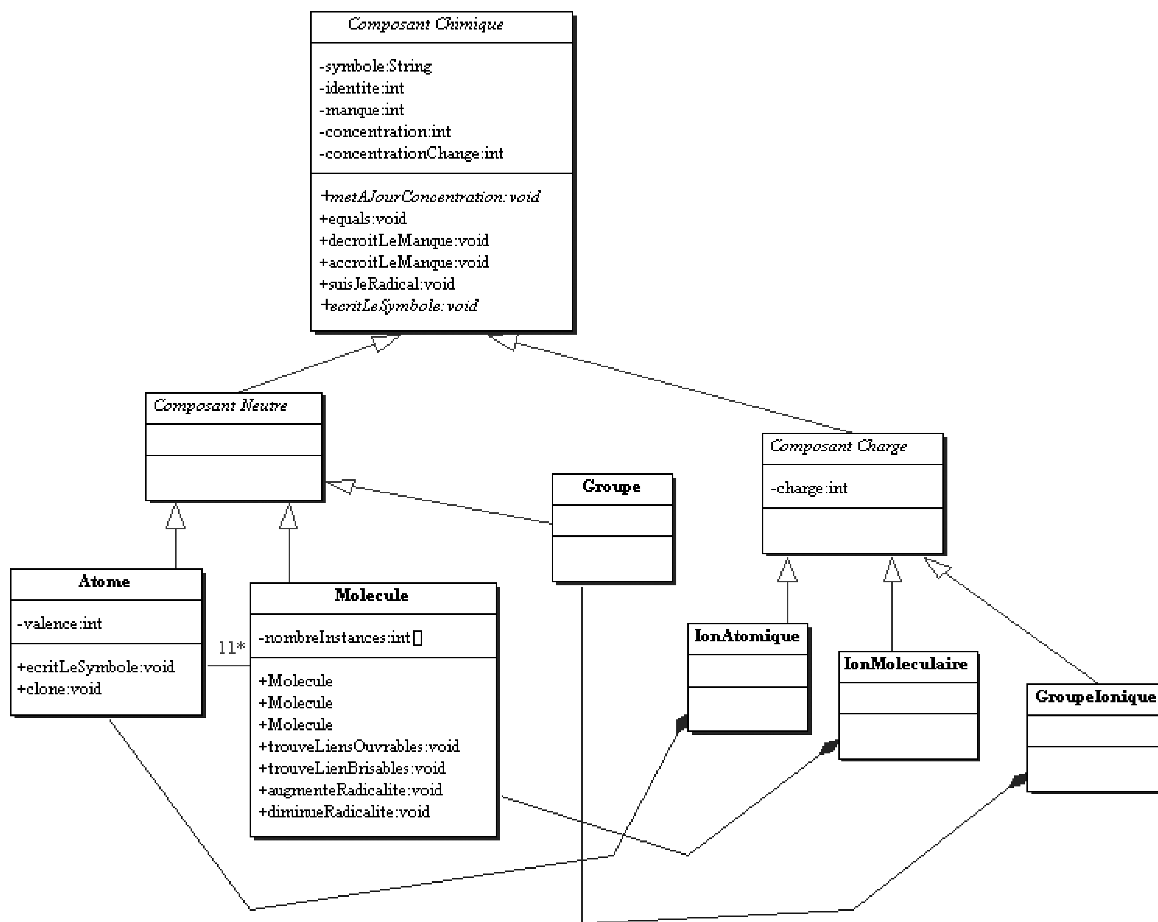


Figure 22-4

Ajout des trois sous-classes de Composant_Charge : IonAtomique, IonMoleculaire et GroupeIonique.

Java rend de toute façon cette solution caduque et lui substitue naturellement une solution pratiquement équivalente : l'utilisation de la relation de composition. Ainsi un objet `IonAtomique` sera-t-il composé d'un objet `Atome` plus une charge additionnelle. Il en va de même pour les deux autres sous-classes. Le nouveau diagramme UML intégrant ces trois nouvelles classes ne devrait poser aucun problème de compréhension. Signalons juste que la redéfinition de la méthode `ecritLeSymbole()` se borne à reprendre le symbole du composant neutre qui lui est associé plus la charge.

La classe NœudAtomique

Nous allons coder les molécules comme des graphes computationnels. Nous avons vu dans le chapitre précédent que la manière la plus simple de coder un graphe est de le structurer comme un réseau de nœuds, chacun des nœuds pointant, à son tour, vers un tableau d'autres nœuds. Parmi les attributs de cette nouvelle classe figurera, bien entendu, l'atome constitutif de ce nœud. La classe `NœudAtomique` possède ainsi un lien d'association 1->1 avec la classe `Atome`.

Elle possède également un lien d'association 1->n avec elle-même. En outre, un `nœudAtomique`, tout comme n'importe quel composant chimique, ce qu'il n'est pas (puisqu'il est sans concentration, possède un « manque » et une « charge ». On supposera que la radicalité de la molécule, ainsi que sa charge, puissent incomber à un `nœudAtomique` particulier. Par ailleurs, chaque nœud est également associé à un tableau de liaisons, qui le connecte aux nœuds suivants dans le graphe.

Dans le simulateur, de nombreuses méthodes composent cette classe. Par simplicité, nous n'en épinglerons que quelques-unes. Par exemple, la méthode `rajouteUneLiaison` rajoutera sous ce nœud un lien au graphe moléculaire. Des méthodes permettent de modifier la radicalité. La méthode la plus importante est sans doute celle qui permet la duplication du nœud : `duplicate`, abondamment surchargée, selon la réaction chimique qui induit cette duplication.

Cette méthode sera appelée quand, à l'issue d'une réaction, la molécule produite sera en partie une réplique de celle de départ. D'autres méthodes sont liées à la « canonisation » de la molécule (que nous discuterons dans la suite et qui permet de réordonner les nœuds vers lesquels pointent chacun d'entre eux), telle la méthode qui permet de découvrir quel est le plus petit d'entre deux nœuds. La classe `NœudMoleculaire` est sans conteste la classe la plus compliquée et la plus riche de notre simulateur, puisqu'elle est entièrement responsable du codage de toute molécule et joue un rôle phare dans le déroulement des réactions.

Chaque molécule possède un `nœudAtomique` que nous appellerons, par la suite, le nœud d'entrée dans le graphe moléculaire.

La classe Liaison

C'est la dernière classe des éléments structurels de notre simulateur. Elle se limite à connecter deux nœuds atomiques ensemble. Elle possède comme premier attribut le nombre de couples électroniques qui la constituent. Ainsi, dans la molécule H_20, les deux liaisons ne possèdent chacune qu'un couple mais, dans la molécule C_2H_2, la liaison entre les deux carbones possède 3 couples. La classe possède enfin deux valeurs d'énergie : d'abord, l'énergie nécessaire pour briser la liaison dans son entièreté, et ensuite l'énergie nécessaire pour ouvrir la liaison, c'est-à-dire pour ne « briser » qu'un des couples qui la constituent.

Comme méthode principale, elle peut accroître son nombre de couples et le décroître. Mais l'essentiel de ses méthodes est relié aux schémas réactionnels auxquels elle participe. Ainsi, la méthode `echangeLiaison` permet d'échanger cette liaison avec une autre, lors d'une réaction que nous désignerons de « croisement ». La méthode `ouvreLiaison` permet à cette liaison d'intégrer deux nouveaux `nœudsAtomiques`, en ouvrant un des couples dont elle est constituée, lors d'une réaction que nous désignerons « d'ouverture de liaison ».

Le diagramme des classes structurelles de notre simulateur chimique est représenté ci-après dans sa version finale.

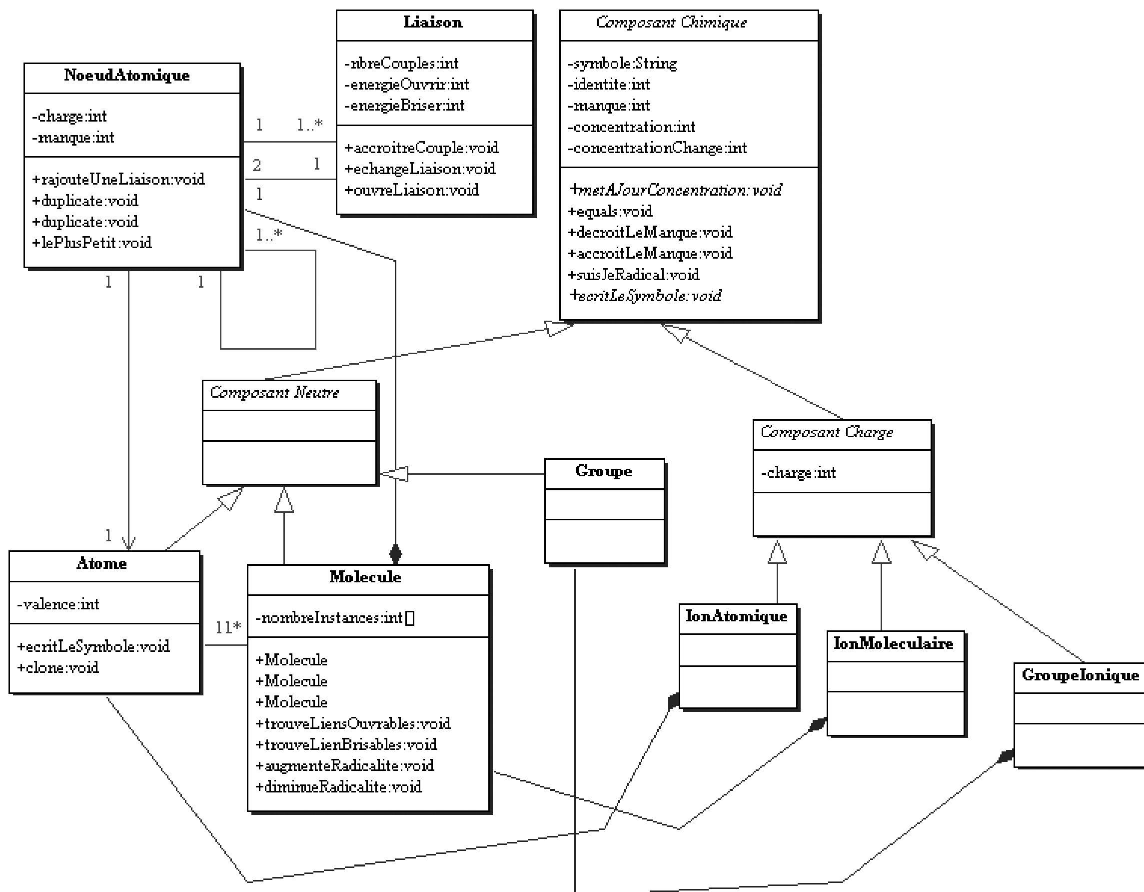


Figure 22-5

Version finale du diagramme de classe des éléments structurels de notre réacteur chimique.

Le graphe moléculaire

Observez la molécule ci-après, il s'agit du butane : C_4H_{10}. Cette molécule est composée de 4 atomes de carbone et de 10 atomes d'hydrogène. On comprend pourquoi la représentation informatique la plus naturelle de cette molécule se fait *via* un graphe. Pour des raisons que nous allons vous préciser tout de suite, la manière dont cette molécule est codée dans notre simulateur est la suivante (rappelez-vous que le symbole du carbone pour nous est « 1 », et celui de l'hydrogène « 4 »).

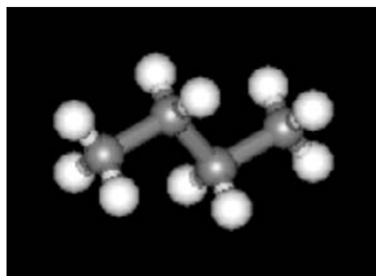


Figure 22-6
La molécule de butane.

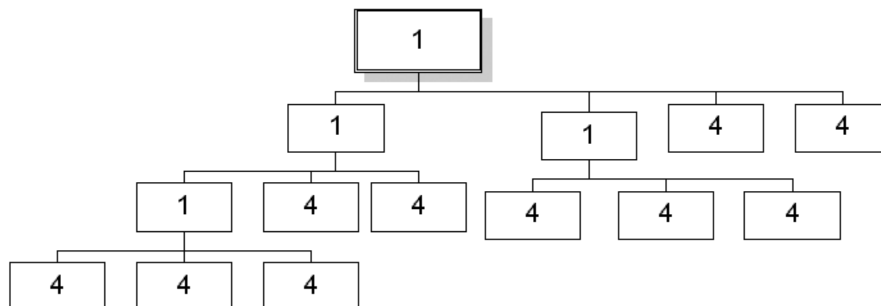


Figure 22-7
Le graphe canonisé du butane.

Le symbole associé à cette molécule est exactement identique à la représentation sous forme de graphe. Il s'agit de : 1(1 (1 (4 4 4) 4 4) 1 (4 4 4) 4 4). On retrouve bien les 4 carbones et les 10 hydrogènes. On constate également que les valences sont bien respectées. Chacun des « 1 » est connecté à 4 autres nœuds atomiques, et chacun des « 4 » à un seul. Toutefois, il est capital ici de comprendre que la structure en graphe de la molécule n'est pas aléatoire. Ce graphe est parfaitement ordonné, on dira « canonisé », afin que les symétries pouvant exister dans la façon dont les atomes sont connectés entre eux laissent la molécule inchangée.

Pour le dire encore plus simplement, pour nous les molécules NaOH et HONa (que l'hydrogène soit à droite ou à gauche de l'oxygène) seront une et une même molécule. Nous exigeons qu'un graphe ne représente qu'une et une seule molécule, quel que soit l'ordre dans lequel les atomes s'y sont connectés, au fur et à mesure de leur arrivée. Ici, nous avons fait la supposition, qui fera bondir plus d'un chimiste, mais qui laissera indifférent la grosse majorité des programmeurs OO (ce qui nous importe le plus ici), que l'ordre dans lequel plusieurs atomes sont connectés à un autre atome n'a aucune espèce d'importance.

En substance, cela revient à négliger complètement la disposition spatiale des atomes dans la molécule, et à ne considérer comme seulement déterminant que la structure de connectivité des atomes entre eux. Énoncée de manière plus chimique, cette représentation sous forme de graphe nous permet de différencier des « isomères structuraux », c'est-à-dire des molécules qui contiennent les mêmes atomes mais dont le graphe de connectivité diffère (ainsi, la molécule méthylpropane est un isomère structurel de la molécule butane).

En revanche, elle ne nous permet pas de différencier des « isomères géométriques », c'est-à-dire des molécules dont le graphe de connectivité est exactement le même, mais dont la disposition géométrique des atomes est différente. Cette différenciation est pourtant fondamentale en chimie, car elle induit de nombreux effets dans les schémas réactionnels qui concernent ces molécules (et que nous négligeons donc complètement ici).

Les règles de canonisation

Voici les deux règles de canonisation que tous nos graphes moléculaires respectent :

- Première règle de canonisation : en dessous de tout nœud atomique, les nœuds atomiques sont ordonnés de manière croissante : du plus petit au plus grand. « En dessous » est un terme ici quelque peu trompeur ; il induit une disposition géométrique qui n'a pourtant aucune importance. Nos molécules sont des graphes, et cette règle concerne en fait tout nœud atomique et tous ceux qui lui sont connectés. Ce sont ces derniers qui s'ordonneront de manière croissante.
- Seconde règle de canonisation : celui que nous avons appelé plus tôt dans le texte le nœud d'entrée du graphe, en fait le nœud de départ de la molécule, sera le plus petit nœud du graphe.

Bien évidemment, ces deux règles n'ont de sens que si l'on spécifie ce que l'on entend par le plus petit d'entre deux nœuds atomiques. Rappelez-vous qu'une des méthodes associées à chaque nœud permet de le comparer à tout autre, afin de savoir lequel des deux est le plus petit.

Déterminer le plus petit des deux nœuds n'est pas une mince affaire, et l'exemple de la molécule de butane vous le fera sans doute mieux appréhender. Nous nous limiterons ici à indiquer les éléments principaux de cette comparaison. Lorsque deux nœuds atomiques concernent deux atomes d'identité différente, comme « 1 » et « 4 », il n'y aucun problème à déterminer le plus petit (d'où l'expression de cette identité comme un entier). Et comme vous le constaterez, dans la molécule de butane, les « 4 » sont systématiquement à droite des « 1 ».

Tout cela se corse sévèrement, lorsqu'il s'agit cette fois de comparer deux nœuds atomiques dont les atomes correspondants ont la même identité, comme c'est le cas quand il s'agit d'ordonner les « 1 » dans notre molécule de butane. Nous nous limiterons à dire que cette comparaison se fait de manière récursive et en largeur, et observerons d'un peu plus près notre molécule de butane.

Prenons par exemple les deux premiers « 1 » sous le « 1 » principal ; le second est plus grand que le premier car, parmi les nœuds atomiques qui lui sont connectés, il y a un « 1 » et trois « 4 », alors que, pour le premier, il y a deux « 1 » et deux « 4 ». Bien sûr, cette comparaison peut entraîner un voyage récursif dans le graphe, voyage se déroulant de nœud à nœud et de niveau à niveau. La manière dont nous avons codé le graphe, un nœud pointant sur un vecteur de nœuds, facilite grandement ce type de circulation récursive. Les programmeurs apprécieront.

La seconde règle de canonisation est également bien respectée, car le premier « 1 » du graphe, le nœud d'entrée, est en effet le plus petit de tous les nœuds, vu qu'il se connecte à deux « 1 » et deux « 4 ». Deux candidats parfaitement identiques étaient possibles pour occuper cette position stratégique, le choix se fait arbitrairement.

Finalement, les molécules sont bien des graphes et non des arbres computationnels, dans la mesure où des cycles peuvent se rencontrer dans ces graphes, comme le fameux benzène C_6H_6, dans lequel le dernier carbone se reconnecte au premier. Dans notre simulation, la notation symbolique du benzène sera comme suit : 1[1](1(1(1(1(1[1](4)4)4)4)4)4)4). La présence additionnelle du [1] dans cette notation permet de signaler les deux carbones qui se relient dans le graphe.

Si une deuxième fermeture se produit dans le graphe, il faudra, cette fois par la présence d'un [2], repérer les nœuds atomiques que cette fermeture concerne, et ainsi de suite, incrémentant le nombre se trouvant entre crochets. Le graphe du benzène est représenté ci-après. Rien de bien particulier ne différencie les graphes des arbres dans la manière dont sont codées nos molécules. La présence de ces crochets n'est due qu'à la volonté de traduire le graphe dans une notation symbolique linéaire.

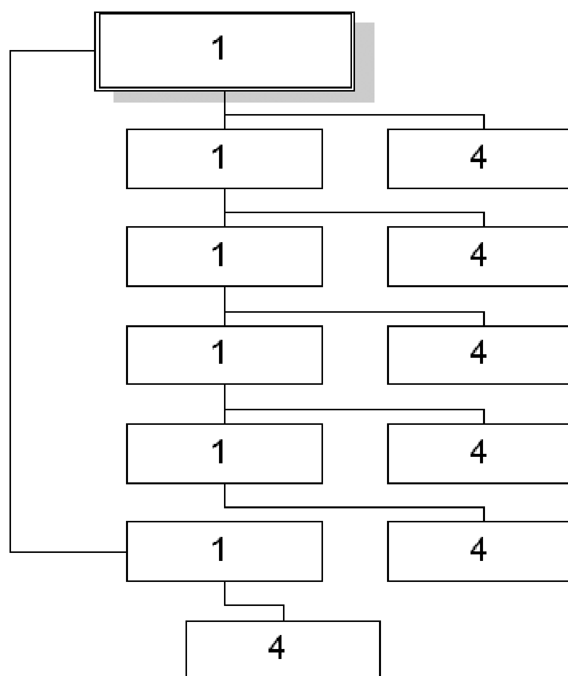
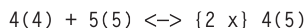


Figure 22-8
Le graphe canonisé du benzène.

Les réactions chimiques

Passons maintenant au deuxième assemblage, comprenant les classes responsables des réactions. Une vingtaine de réactions sont codées dans le simulateur. Avant de décrire plus en détail de quoi est faite la superclasse `Reaction`, décrivons quelques réactions chimiques typiques. Commençons par une réaction de « croisement » élémentaire :

Une première réaction de croisement



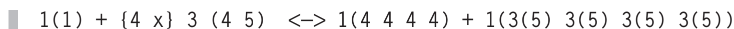
```
4(4) + 5(5) <–> {2 x} 4(5)
```

Les deux molécules de départ sont des molécules diatomiques, comprenant deux atomes, et chaque atome n'est connecté à l'autre que par une liaison à un seul couple électronique. Cette réaction se produit par rupture des deux liaisons des molécules de départ et par croisement des atomes. Un exemple chimique de ce type de réaction serait : $H_2 + Cl_2 <–> 2 HCl$. Du fait de la canonisation des molécules 5(4) devient 4(5), avec pour conséquence la présence du { 2 x }, que les chimistes appellent un coefficient stœchiométrique, et qui permet de s'assurer que le même nombre d'atomes se retrouve bien à droite et à gauche de la réaction.

Vous aurez remarqué que la flèche reliant les deux parties de la réaction va dans les deux sens. En effet, les chimistes s'accordent pour dire que toutes les réactions sont symétriques, bien qu'en règle générale, elles vont beaucoup plus vite dans un sens que dans l'autre. Elles vont bien plus vite quand elles sont exothermiques et qu'elles libèrent de l'énergie, les molécules produites étant plus stables que les molécules réactantes. Néanmoins, en absorbant suffisamment d'énergie, elles peuvent se produire, plus laborieusement toutefois, en délivrant des molécules à l'arrivée plus énergétiques que les molécules de départ. De nouveau, notre description des

processus est volontairement simplifiée, nous en excusant à l'avance auprès de ses praticiens. Toutes les réactions de notre simulateur seront en effet symétriques.

Une autre réaction de croisement un peu plus sophistiquée :



```
1(1) + {4 x} 3 (4 5)  <–> 1(4 4 4 4) + 1(3(5) 3(5) 3(5) 3(5))
```

Ici, la liaison 1-1 (constituée de 4 couples) et la liaison 3-4 (constituée de 1 couple) se brisent toutes deux, pour former les deux nouvelles molécules. La présence du {4 x} est due à la valence 4 de l'atome « 1 ». Poursuivons maintenant par un deuxième type de réaction, « ouverture de liaison ».

Une réaction d'ouverture de liaison :

```
1(3 3) + 3(4 4) <–> 1(3(4) 3(4) 3)
```

On s'aperçoit qu'au contraire des réactions précédentes, deux réactants ne conduisent ici qu'à un seul produit. Cette réaction se produit car une des deux liaisons 1-3 (constituée de deux couples, « 1 » est de valence 4 et « 3 » est de valence 2) s'ouvre, c'est-à-dire libère un de ses couples, tandis qu'une des deux liaisons 3-4 (constituée d'un seul couple) de la seconde molécule se brise. Le « 4 » ira d'un côté de la liaison, le « 3-4 » de l'autre. La molécule résultante est bien canonisée car, entre deux nœuds atomiques de même identité atomique, le plus petit sera celui le plus fortement connecté. Ici, les deux premiers « 3 » ont deux connexions chacun, alors que le dernier n'en a qu'une.

Deux exemples de réaction de « croisement » et « d'ouverture de liaison » sont représentés dans les figures ci-après :

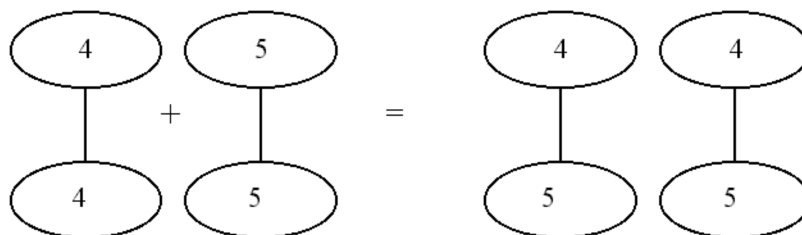


Figure 22-9
Illustration de la réaction de croisement : 4(4) + 5(5) <–> {2 x} 4(5).

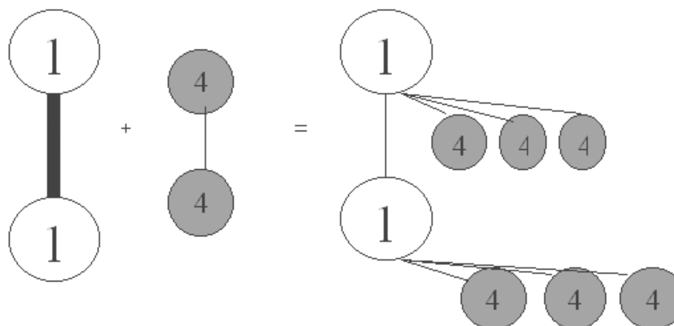
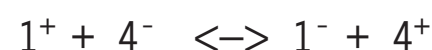


Figure 22-10
Illustration de la réaction d'ouverture de liaison : 1(1) + {3 x} 4(4) <–> 1(1(4 4 4) 4 4 4).

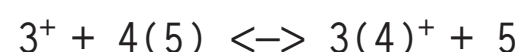
Bien d'autres schémas réactionnels ont été prévus dans le simulateur, y compris des réactions impliquant la nature ionique des molécules, comme la simple réaction suivante de transfert de charge :

Réaction de transfert de charge :



$1^{+} + 4^{-} <-> 1^{-} + 4^{+}$

Ou encore des réactions de *type IonMolecule* comme :



$3^{+} + 4(5) <-> 3(4)^{+} + 5$

Comment est calculée la cinétique réactionnelle

Quand une réaction de type A + B -> C + D se produit, la concentration des molécules A et B va décroître, alors que celle des molécules C et D va croître d'autant. Dans notre simulateur, la vitesse réactionnelle K_{ABCD}, en conformité avec la théorie cinétique de la chimie, est calculée de la manière suivante :

$$K_{ABCD} = \alpha \exp(-E_{ABCD}/\beta T)$$

α et β sont deux paramètres qui seront des attributs des objets réactionnels. Leur signification est liée à l'orientation et la forme des molécules rentrant dans la réaction. T est la température à laquelle se déroule la réaction. Dans notre simulateur, les valeurs de ces paramètres seront fixées arbitrairement.

E_{ABCD} est la valeur d'énergie d'activation de la réaction. C'est cette valeur qui est responsable du sens effectif de la réaction. En effet, comme la dépendance à cette valeur est exponentielle, si celle-ci est grande (la réaction doit alors franchir une grande barrière énergétique pour se réaliser), la réaction ne se produira quasiment pas. En revanche, si cette valeur est petite (la barrière énergétique à franchir est plus faible), la réaction se produira plus facilement et rapidement.

Nous calculerons E_{ABCD} de la manière qui suit. De nouveau, la démarche est quelque peu naïve en regard de la chimie, mais notre programmeur OO s'en satisfera. Chaque liaison de la molécule qui s'ouvre ou se brise a une certaine énergie d'ouverture ou rupture qui lui est associée. C'est, en effet, l'énergie nécessaire pour ouvrir ou briser cette liaison.

```
EABCD = (Σ energies des liaisons des réactants – Σ énergies  des liaisons des
produits) + Δ    si la difference est positive
EABCD = Δ         si la difference est négative ou nulle
```

Δ est un paramètre arbitraire, fixé dans le simulateur. La différence s'opère entre la somme des énergies des liaisons brisées ou ouvertes dans les molécules de départ et la somme des énergies des liaisons ré-établies dans les molécules à l'arrivée. De la sorte, une réaction menant à des molécules plus stables (cette différence est négative) devra absorber beaucoup moins d'énergie, juste la barrière énergétique Δ, et se déroulera bien plus vite. En effet, la vitesse cinétique de l'évolution des concentrations est calculée comme suit :

```
d[A]/dt = d[B]/dt = - KABCD[A][B]
d[C]/dt = d[D]/dt = KABDC[A][B]
```

Où la présence des crochets indique la concentration des molécules concernées. Plus E_{ABCD} est petit plus K_{ABDC} est grand, et plus la réaction se déroule rapidement.

La classe Reaction

Comme vous le constaterez à partir du diagramme de classe qui va suivre, les principaux attributs de la classe `Reaction` résultent de ses relations avec la classe `Composant Chimique` (chaque réaction est associée avec au plus quatre de ces composants, neutres ou ioniques, moléculaires ou atomiques) et la classe `Liaison` (chaque réaction est associée avec au plus quatre liaisons, celles qui se brisent dans les molécules de départ et se

ré-établissent à l'arrivée). La classe possède également deux attributs réels stockant les vitesses réactionnelles dans un sens et dans l'autre. La classe `Reaction` est une superclasse abstraite, qui donnera naissance à un ensemble de sous-classes concrètes, consistant en tous les mécanismes réactionnels simulés dans notre réacteur.

Parmi les nombreuses méthodes qui constituent cette classe, nous en considérons trois. La méthode `etablit-LaReaction()` est une méthode abstraite, mais dont le rôle est de créer la réaction, dans le sens de découvrir les molécules produites, de les canoniser, de vérifier si ces molécules existent déjà ou pas, et, finalement, de calculer à partir de ces dernières les vitesses réactionnelles. La méthode `executeUnPas()`, elle aussi abstraite, considère que la réaction est connue, et se limite à exécuter un pas de temps de cette réaction, c'est-à-dire de simplement décroître la concentration des réactants et d'accroître la concentration des produits selon les vitesses réactionnelles.

On voit poindre la pratique du polymorphisme, car les deux messages correspondant aux deux méthodes décrites seront envoyés à tour de rôle sur un vecteur d'objets réactionnels, avec des effets différents selon la nature précise de la réaction. La méthode `equals()` est également redéfinie pour les réactions, et permet de comparer deux réactions. Deux réactions seront en effet égales si, à partir des mêmes réactants, elles produisent les mêmes produits.

Les sous-classes de Reaction

Une vingtaine de sous-classes concrètes de réaction héritent de la classe `Reaction`, comme les classes `Reac-tion Croisement`, `Reaction Ouverture Liaison`, `Reaction TransfertCharge`, `Reaction IonMolecule`. Pour chacune de ces réactions sont redéfinies les méthodes `etablitLaReaction` et `executeUnPas`. Le diagramme UML qui suit ne reprend, par souci de clarté, qu'une partie du diagramme UML précédent, en lui rajoutant les classes chargées des réactions.

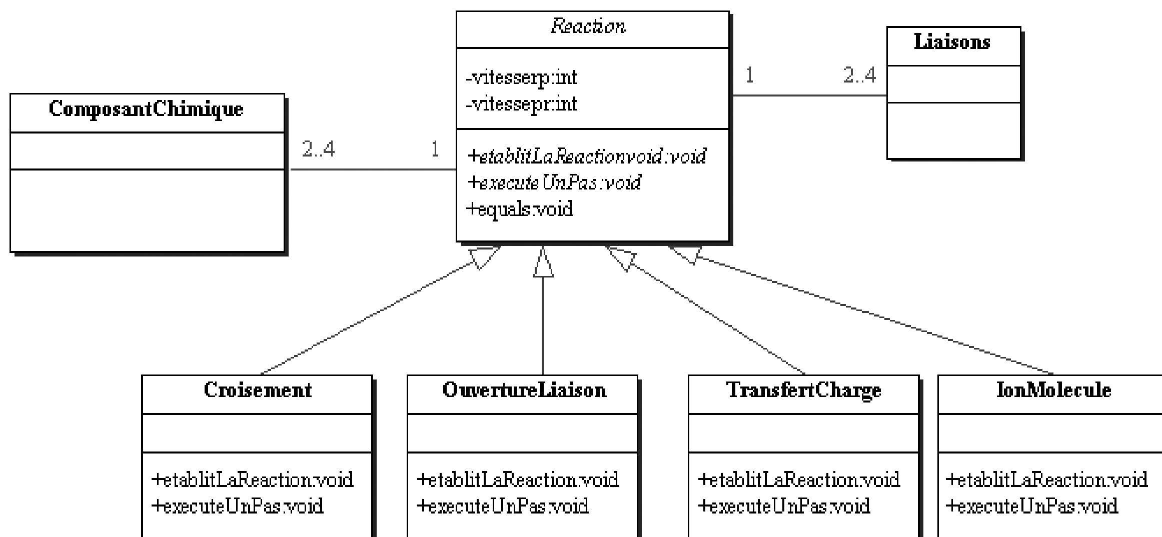


Figure 22-11

Partie du diagramme de classe UML indiquant les classes Réactions.

Quelques résultats du simulateur

Dans le simulateur, trois vecteurs d'objets sont importants. Le premier vecteur contient tous les composants chimiques existant dans le réacteur. Le deuxième vecteur renferme seulement les composants chimiques apparus lors de la dernière étape de la simulation. Cette séparation entre ces deux vecteurs permet de s'assurer que les prochaines réactions ne se produisent qu'entre tous les éléments du premier vecteur et ceux du deuxième. Le troisième vecteur comprend tous les objets réactionnels.

À chaque étape de la simulation, on s'emploie d'abord à créer de nouvelles réactions, en essayant les différents schémas réactionnels possibles entre les composants du premier vecteur et ceux du deuxième. Enfin, on exécute un pas de simulation sur tous les objets réactionnels déjà existants. Une explosion combinatoire de nouvelles molécules se produit si on n'y prend pas garde. Nous proposons dans le simulateur plusieurs manières de l'éviter, par exemple en n'autorisant que les molécules suffisamment concentrées à participer aux réactions. De surcroît, les réactions se limitent à ne briser que les liaisons les plus faibles (c'est pour cette raison qu'on les mémorise dans la classe `Molecule`).

De manière à vous convaincre que ce procédé ne génère pas des résultats aussi étranges qu'on pourrait initialement le craindre, nous allons montrer les résultats obtenus sur trois simulations, de complexité croissante, en démarrant ces simulations à partir de molécules initiales différentes.

La première simulation démarre à partir des deux molécules 4(4) et 5(5). Une seule nouvelle molécule est créée. La figure qui suit montre l'évolution des concentrations des trois molécules concernées. La réaction est largement exothermique, et la nouvelle molécule prend vite le pas sur les deux autres.

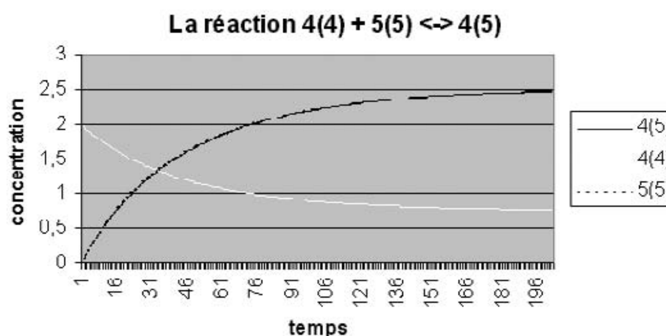


Figure 22-12
Évolution des concentrations des trois molécules participant à une réaction de croisement.

La deuxième simulation démarre à partir des deux molécules 2(2) et 4(4). Vu que l'atome « 2 » est de valence 3, une flopée de nouvelles molécules apparaissent. Par exemple :

```
2 ( 4 4 4 )
2 ( 2 ( 4 ) 4 )
2 ( 2 ( 4 4 ) 2 ( 4 4 ) 2 ( 4 4 ) )
2 ( 2 ( 2 ( 4 ) ) 4 4 )
2 ( 2 ( 4 4 ) 4 4 )
2 ( 2 ( 4 4 ) 2 ( 4 4 ) 4 )
2 ( 2 ( 2 ( 4 ) ) 2 ( 2 ( 4 ) ) 2 ( 2 ( 4 ) ) )
2 ( 2 ( 2 ( 4 ) ) 2 ( 4 ) )
2 ( 2 ( 2 ( 4 4 ) 4 ) 2 ( 4 4 ) 4 )
2 ( 2 ( 2 ( 2 ( 4 4 ) ) ) 4 4 )
```

Le profil de concentration est montré dans la figure qui suit :

Figure 22-13
Évolution des concentrations des molécules issues des deux molécules de départ : 2(2) et 4(4).

Nous mettons au défi tout chimiste d'arriver à prévoir, sans recourir à ce type de simulation, comment évolueront ces concentrations et quelles seront les molécules les plus concentrées. On a toujours besoin de petits informaticiens chez soi.

Finalement, dans une troisième simulation, les molécules de départ sont les suivantes :

```
1[1](1(1(1(1(1[1](4)4)4)4)4)4)  et 5(5)
```

La première molécule est la molécule cyclique du benzène. Là encore, une multitude de nouvelles molécules apparaissent, des molécules organiques bien connues des chimistes. Certaines de ces molécules commencent à être d'une certaine complexité, comme vous pouvez le voir. Il en va ainsi de la dernière, qui présente deux fermetures dans son graphe.

```
1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 ) 4 )
4 ) 4 ) 4 ) 4 ) 4 ) 4 )

 1 ( 1 ( 1 ( 1 ( 4 5 ) 4 ) 4 ) 1 ( 1 ( 4 5 ) 4 ) 4 )

 1[1] ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 5 ) 4 5 )

 1[1] ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 )

 1 ( 1 ( 4 ) 4 )

1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1[1] (
4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 )

 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 4 5 ) 4 ) 4 ) 4 ) 4 ) 4 ) 1 ( 1 ( 1 ( 1 ( 1 ( 4
5 ) 4 ) 4 ) 4 ) 4 ) 4 )

 1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 ) 4
) 4 ) 4 ) 4 ) 4 ) 4 5 ) 4 5 )
```

```
 1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1
( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 )
4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 )

 1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4
) 4 ) 4 )

 1[1] ( 1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 )

 1[1] ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 5 ) 1 ( 1 ( 1 ( 1 ( 1 ( 1 (
1 ( 1 ( 1 ( 1 ( 1 ( 1 ( 4 5 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 ) 4 )

1[1] ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 5 ) 4 ) 4 ) 4 5 ) 4 5 ) 4 5 )

1[2] ( 1[1] ( 1 ( 1 ( 1 ( 1 ( 1[1] ( 4 ) 4 ) 4 ) 4 ) 4 5 ) 4 ) 1 ( 1 ( 1 ( 1
( 1[2] ( 4 ) 4 ) 4 ) 4 ) 4 5 ) 4 )
```

Le rêve ultime qui sous-tend l'expérimentation de ce simulateur serait de pouvoir un jour reproduire la fameuse expérience de Miller qui, à partir de quelques molécules élémentaires et de quelques stimulations énergétiques (l'idée était de reproduire dans son réacteur une espèce de soupe pré-biotique et une atmosphère primitive), avait découvert, surpris, en ouvrant sa boîte de Pandore, des molécules étonnamment complexes et de premiers acides aminés. On était encore à mille lieues de l'ADN, mais plus d'un chimiste et plus d'une biologiste s'étaient émus de découvrir la complexité atteinte par ce processus pourtant élémentaire et aléatoire, comme si toute la richesse du vivant pouvait s'obtenir, au départ, à partir de trois fois rien.

La simulation immunologique en OO ?

Petite introduction au fonctionnement du système immunitaire

Lorsqu'un virus, microbe ou autre nuisance (que les immunologistes généralisent sour le nom « d'antigène ») s'égare à l'intérieur de nos organismes, toute une batterie de défense cellulaire se met en branle afin de contrer et éliminer l'intrus. Au risque de simplifier abusivement (le but étant de sensibiliser le lecteur à l'adéquation de l'OO pour la biologie plus qu'à la biologie elle-même), voici quelques éléments descriptifs très introductifs. Les cellules immunitaires, principalement les lymphocites T ou B, déploient différentes stratégies de défense : certaines (les lymphocites B) produisent des anticorps qui s'accrochent à l'intrus afin de le neutraliser, alors que d'autres (les lymphocites T) se chargent directement de cette élimination. Il faut pour cela qu'elles aient détecté et « reconnu » l'antigène. Afin de reconnaître l'intrus et le neutraliser, les lymphocites nécessitent un ensemble d'autres types cellulaires pour les assister dans leur tâche. Par exemple, l'antigène devra être présenté aux lymphocites en question par des cellules dites « présentatrices d'antigène » afin que ces lymphocites passent de l'état « naïf » à un état « excité », et déclenchent leur réponse à l'intrus. Ces lymphocites requièrent souvent la présence d'un deuxième signal, envoyé par un autre type cellulaire, en vue de déclencher cette réponse. Enfin, une fois l'élimination accomplie, les cellules qui en sont responsables rentrent dans un nouvel état dit « mémoire », qui leur permet de subsister plutôt que disparaître naturellement. Elles peuvent ainsi assurer une nouvelle réponse, encore plus intense, si d'aventure le même antigène se représentait (c'est le principe des vaccins : ils permettent de générer ces cellules mémoires à partir d'un antigène – mis hors d'état de nuire – injecté).

L'immense diversité des acteurs cellulaires et la succession des différents états dans lequel peut se trouver chacun, selon qu'il rencontre ou non l'antigène ou qu'il soit sollicité ou non par un partenaire cellulaire, rendent indispensable un mode de programmation capable de facilement intégrer et maîtriser cette complexité. L'OO devrait, une fois encore, vous apparaître comme la voie naturelle de prise en compte informatique de ce secteur

des sciences. Basons-nous, par exemple, sur deux petits extraits d'un traité d'immunologie de Janeway, devenu un ouvrage classique que tous les étudiants en médecine ou biologie ont sans doute rencontré au moins une fois dans leur vie :

« L'activation due à la rencontre de l'antigène avec les cellules effectrices T est assistée par des co-récepteurs présents sur les surface de ces cellules T. Ces récepteurs sont capables de faire la différence entre deux classes de molécules MHC : les cellules T cytotoxiques expriment sur leur surface le co-récepteur CD8 qui se lie au molécule MHC de classe I tandis que les cellules T qui portent sur leur surface le co-récepteur CD4 ne peuvent reconnaître que la molécule MHC de classe II. »

Il serait si simple, et éventuellement plus clair, de remplacer ce texte par le diagramme de classe ci-dessous, dont l'objectif est d'exprimer exactement la même connaissance mais cette fois à l'aide d'un petit dessin.

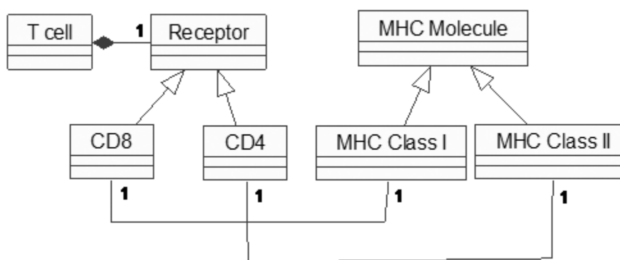


Figure 22-14
Traduction dans un diagramme de classe du premier extrait d'immunologie

Voici un autre extrait :

« Les cellules T sont activées et se transforment en cellules T effectrices lorsqu'elles rencontrent leur antigène spécifique sous forme d'un peptide qui leur est présenté par la molécule MHC installée sur la surface d'une cellule présentatrice d'antigène activée (APC). Les plus importantes APC sont les très spécialisées cellules dentritiques… Des macrophages peuvent également être activés afin d'exprimer sur leur surface l'antigène, mais à partir cette fois des molécules MHC de classe II… Les cellules lymphocites B peuvent également jouer le rôle d'APC dans certaines circonstances. »

Un diagramme de classe pourrait à nouveau faire l'affaire pour traduire cette connaissance sous une forme plus synthétique.

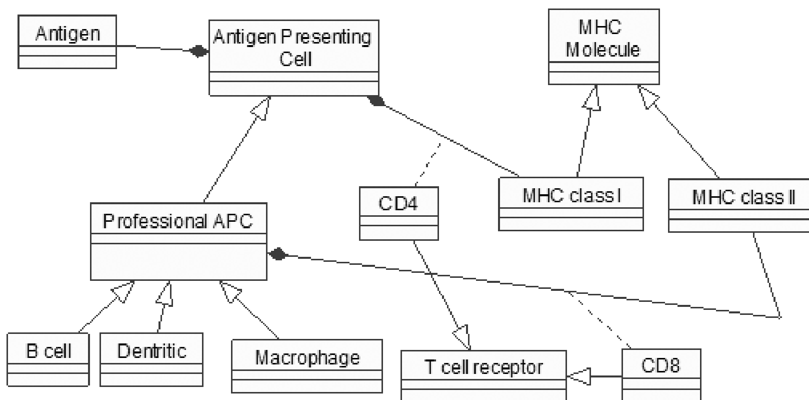


Figure 22-15
Traduction dans un diagramme de classe du deuxième extrait d'immunologie

Le diagramme UML d'état-transition

Au-delà du diagramme de classe ou de séquence déjà rencontré dans les chapitres précédents, il existe un autre diagramme UML, bien utile, vous permettant de suivre dans le temps l'évolution d'un objet, lorsque celui-ci est sujet à des états variés et à différentes transitions entre ces états. La figure 22-16 permet de comprendre facilement les éléments graphiques principaux qui composent le diagramme d'état-transition. Le disque noir débute le diagramme. Lorsque le code s'exécute, il signifie la naissance de l'objet en question. Le disque noir entouré de blanc signifie lui, au contraire, la disparition de l'objet. Tous les états dans lesquels l'objet peut se trouver sont représentés par des rectangles aux bords arrondis. Les transitions font le liant entre les états dans la mesure où elles permettent de passer d'un état à l'autre. Cela peut se produire suite à la réception d'un message par l'objet en question ou comme conséquence de la logique propre d'évolution d'un objet (comme le départ à la retraite d'une personne à partir d'un certain âge). La figure 22-16, qui représente très sommairement les différentes étapes possibles de la vie professionnelle d'une personne, devrait vous permettre de facilement saisir le rôle et les éléments constitutifs de ce diagramme.

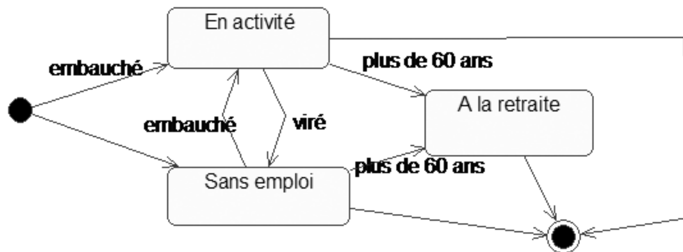


Figure 22-16
Diagramme d'état-transition de la vie professionnelle d'un objet Personne

Voici maintenant le diagramme d'état-transition d'une cellule immunitaire responsable de nos défenses organiques, T ou B. Il résume assez bien ce que tout immunologiste sait du fonctionnement du système qu'il étudie.

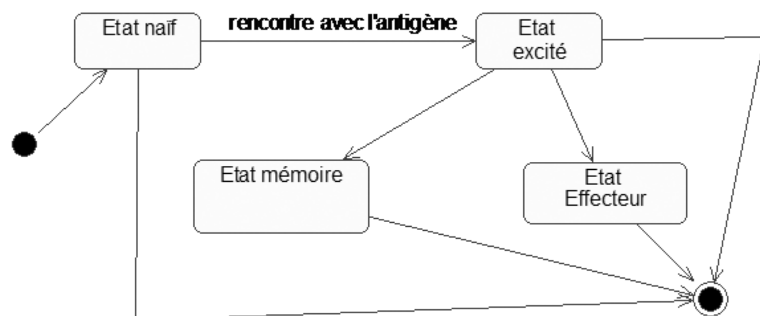


Figure 22-17
Diagramme d'état-transition d'une cellule immuniatre T ou B dès qu'elle rencontre un antigène

Dès que l'antigène est rencontré, la cellule immunitaire se place d'abord dans un premier état d'excitation. À partir de cet état, le diagramme indique qu'elle pourra se transformer soit en une cellule effectrice, soit en une cellule mémoire. En tant que cellule effectrice, elle aura pour tâche d'éliminer d'une manière ou d'une autre l'antigène qu'elle a rencontré. En tant que cellule mémoire, elle modifiera son horloge interne de sorte à ne pas disparaître trop rapidement. La manière la plus directe de traduire cette réalité dans un code, Java par exemple, est de recourir au design pattern d'état (sur lequel nous reviendrons dans le prochain chapitre).

Celui-ci exige, comme la figure l'illustre, de créer une superclasse abstraite ou une interface `Etat` et autant de sous-classes qu'il n'y a d'état possible. Chaque sous-classe etat a la responsabilité de coder ce qui se passe tant que la cellule se trouve dans cet état et notamment le type de transition possible à partir de cet état.

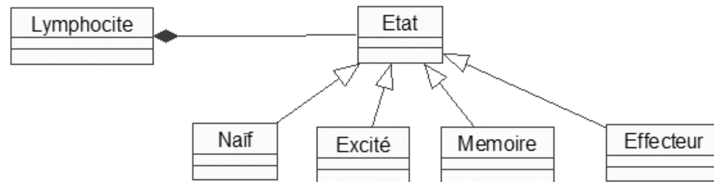


Figure 22-18
Diagramme de classe associé au diagramme d'état-transition de la figure 22-17

Une partie très simplifiée du code Java (limité à une seule sous-classe d'état) associé à ces deux diagrammes UML serait plus ou moins semblable au code suivant :

```
class TCell
{
       Recepteur myRecepteur;
       Etat etat_naif;
       Etat etat_excite;
       Etat etat_memoire;
       Etat etat_effecteur;

       Etat etat_courant;

       public TCell()
       {
             // les différents états de la cellule
             etat_naif = new E_Naif(this);
             etat_excite = new E_Excite(this);
             etat_memoire = new E_Memoire(this);
             etat_effecteur = new E_Effecteur(this);

             etat_courant = etat_naif; // l'état courant

       }

       // les différentes fonctions get pour chaque état

       public String getEtat(){
             return etat_courant.val_Etat();
       }

       public void setEtat(Etat val){
             etat_courant = val;
       }

       public Etat getetat_courant(){
             return etat_courant;
       }

       public Etat getE_Naif(){
             return etat_naif;
       }
```

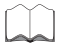
```
        public Etat getE_Excite(){
              return etat_excite;
        }

        public Etat getE_Effecteur(){
              return etat_effecteur;
        }

        public Etat getE_Memoire(){
              return etat_memoire;
        }

        public void simulation() {
              etat_courant.transition();
        }

}

interface Etat {
        public String val_Etat();
        public void transition();
}

class E_Naif implements Etat
{
        TCell tcell;

        public E_Naif(TCell t)   {
              this.tcell = t;
        }

        public void transition(){
              tcell.setEtat(tcell.getE_Excite());

        }

        public String val_Etat(){
              return "Naif";
        }
}
```

23

Design patterns

Les design patterns sont des recettes de conception OO qui à l'origine furent réunies dans le célèbre livre du gang des quatre. Ils sont depuis devenus presque aussi courants que les langages de programmation et UML. Leur connaissance complète toute formation approfondie en OO et permet une meilleure compréhension des mécanismes de ce style de programmation. Ce chapitre en décrit quelques-uns à l'aide d'exemples concrets.

Sommaire : Introduction aux design patterns — Leur côté « truc et ficelle » — L'approfondissement des mécanismes OO



Candidus — À supposer que je rencontre un problème de conception lors d'un développement logiciel quelconque, connais-tu une bonne référence dans laquelle les problèmes types seraient décrits, ainsi que les meilleures façons de les résoudre ?

Doctus — Et comment ! Cet ouvrage est aussi célèbre dans la communauté OO que tous les langages qui y sont utilisés. Il s'agit du livre sur les design patterns écrit par quatre experts de la programmation OO. Il a depuis été complété par une pléthore de livres.

Cand. — Est-il accessible au premier venu, moi en l'occurrence ?

Doc. — Malheureusement, pas tout à fait. En effet, les design patterns sont des solutions très sophistiquées, qui répondent à des problèmes qui, bien souvent, ne le sont pas moins. En conséquence, ils requièrent que le développeur ait au préalable bien cerné la nature du problème, et qu'il ait la certitude de rencontrer ledit problème.

Cand. — Tu veux dire qu'il faut d'abord avoir compris, si mon moteur a soudain des ratés, qu'il s'agit d'un problème d'alimentation d'essence avant de trouver le « truc et ficelle » qui permettra de le réparer ?

Doc. — En gros, oui. Je te l'accorde, le diagnostic est souvent beaucoup plus ardu que la mise en œuvre du traitement ; c'est la difficulté inhérente aux design patterns. Remarque toutefois que les problèmes qu'ils permettent de résoudre sont inéluctables dès lors que le projet atteint une certaine complexité, qu'il fait appel à de nombreuses possibilités d'héritage et ouvre quantité de portes sur le polymorphisme.

Cand. — Si je n'ai pas compris le polymorphisme, autant oublier les design patterns, alors ? Les bras m'en tombent !

Doc. — Ramasse-les. Il t’est en effet possible de mettre quelque peu la charue avant les bœufs (avec les bras, c’est plus facile). Tu peux tirer parti des efforts déployés par Alexander, Ishikawa et Silverstein et par Gamma, Helm, Johnson et Vlissides et leurs successeurs pour justifier l’apport du polymorphisme et de l’encapsulation dans un tas de situations, afin de mieux comprendre et de maîtriser ces mécanismes.

Cand. — Cela revient à utiliser ces design patterns comme des outils pédagogiques pour l’OO…

Doc. — C’est un peu le pari que nous faisons dans ce chapitre, en effet.

Cand. — Pour comprendre les problèmes en découvrant leur solution…

Doc. — Personnellement, je mets cela en œuvre tous les jours avec ma femme et mes enfants, objets chers à mon cœur s’il en est. Mais là, je te l’avoue, j’attends avec impatience un ouvrage comparable à celui du gang des quatre à usage familial.

Introduction aux design patterns

De l’architecte à l’archiprogrammeur

En 1977 et 1979, C. Alexander, S. Ishikawa et M. Silverstein publient deux livres, *A Pattern Language : Towns/ Building/Construction* et *The Timeless Way of Building*, destinés aux architectes et proposant des solutions génériques à des problèmes récurrents en architecture. Ce qui motive ces deux ouvrages, c’est la constatation qu’il existe en architecture – mais cela est vrai dans bien d’autres disciplines, que ce soient la cuisine ou la mécanique automobile – des recettes de conception qui répondent de manière quasi identique à des problèmes aux caractéristiques très semblables. Ces recettes s’avèrent meilleures que d’autres et peuvent en outre participer à la résolution de ces mêmes problèmes. Les réalisations architecturales, que ce soient des basiliques, des mosquées, des salles de concert ou des stades de foot, présentent pas mal d’aspects communs. Ma main à couper, que nos deux maisons existent dans de multiples versions, à commencer par celles de nos voisins…

Dans toute discipline, quelle qu’elle soit, l’expert est celui qui a développé un flair suffisant pour identifier rapidement les problèmes types et qui sait comment les résoudre facilement et vite. Les auteurs de ces deux ouvrages d’architecture sont des archi-experts. Ils ont appelé les recettes de conception objets de leur travail des « design patterns » (par facilité et surtout parce qu’aucune traduction ne fait l’unanimité à ce jour, nous conservons le terme anglais). Le but de leur deux livres est de communiquer ces recettes aux architectes. Pourquoi sans cesse réinventer la roue alors que la solution existe déjà, dans des livres qui ne demandent qu’à vous tracer la voie ?

En 1995, le GOF, Gang Of Four, (voir encadré) décide d’imiter ces précurseurs, dans le domaine du développement logiciel cette fois. Les trois experts architectes ont laissé leur place à quatre experts informaticiens. Il est intéressant de noter combien l’architecture sert de référence au développement logiciel, ce dernier insistant chaque jour davantage sur la nécessité d’une étape préalable de modélisation à l’aide d’un langage formel (UML), avant de se lancer tête dans le guidon, ou plutôt truelle en main, dans la construction du code. Si la programmation spaghetti s’apparente plutôt à du Gaudí, la programmation tordue à du Gehry, ce dont les programmeurs rêvent tous, ce sont de codes structurés à la Niemeyer, dans lesquels il est facile de se retrouver et qu’il est tout aussi simple de faire évoluer. Les trois amigos ont vu dans le langage UML la version informatique des notations que respectent les architectes dans la réalisation de leur plan. Le gang des quatre a transposé la pratique des design patterns architecturaux dans l’univers du logiciel. On attend toujours la contribution du club des Cinq et du clan des Sept.

Si le recours à ces patterns ne semble devoir intervenir qu’une fois le problème rencontré et pas avant, il n’en reste pas moins vrai que leur maîtrise permet de réorienter, et cela dès le début, la conception du code en leur donnant une place prépondérante. Cette réorientation ne peut être que bénéfique puisqu’elle reprend la longue

expérience des nombreux développeurs antérieurs. Certains vont même jusqu'à parler de développement par les « patterns » ou à partir d'eux, se limitant à une simple combinaison de ceux-ci. Déjà, on les retrouve pré-compilés dans un grand nombre d'environnements de développement logiciel qui font également la part belle à UML et au MDA.

Les « Design Patterns » du « gang des quatre » : Gamma, Helm, Johnson et Vlissides

En 1995, un livre intitulé *Design Patterns : Elements of Reusable Object-Oriented Software*, parut chez Addison-Wesley. Il allait marquer considérablement la communauté informatique et son impact perdure jusqu'à aujourd'hui. Ce livre contient vingt-trois solutions architecturales pour les problèmes que tout informaticien rencontre au moins une fois dans sa vie. Pour chaque problème, il n'est pas proposé de code prémâché contenant la solution, mais un « plan de résolution » exprimé dans un langage de modélisation style UML (quelques lignes de code en C++ sont quelquefois ajoutées) qui dévoile les briques structurelles d'une solution élégante. Cet ouvrage de référence, comme tout ouvrage de référence, doit toujours être à portée de votre main, car les solutions qui y sont proposées sont suffisamment originales et subtiles pour ne pas révéler toute leur pertinence à la première lecture. Les quatre auteurs ont compilé, plutôt qu'inventé de toutes pièces, ces solutions génériques, glanées ici et là dans la communauté informatique. Depuis, de nombreux ouvrages sont parus présentant de nouveaux patterns, souvent dédiés à tel ou tel langage de programmation.

Erich Gamma est le directeur technique du Centre de technologie logicielle de Object Technology International (OTI) à Zurich. Il très impliqué dans l'environnement de développement Java, Eclipse et dans le développement de plates-formes logicielles qui favorisent le travail coopératif. Richard Helm appartient à l'Object Technology Practice Group de l'IBM Consulting Group à Sydney en Australie et travaille actuellement dans l'agence de consulting : « Boston Consulting Group ». Ralph Johnson est membre du département informatique de l'université d'Illinois. John Vlissides, disparu en 2005, était chercheur au IBM T.J. Watson Research Center à New York. Ensemble, ils ont formé le « gang des quatre » (*Gang of Four* ou GOF en anglais très « zippé »). Dirigé par Mao Tsé-Toung et par sa femme, avide de pouvoir, la bande des quatre (composée des quatre plus influents dignitaires chinois de l'époque) a dévasté pendant des années la vie politique, économique et culturelle de la Chine. En octobre 1976, un mois après la mort de Mao, les membres du « gang » furent arrêtés et écroués, ainsi que, quelque temps plus tard, la femme de Mao, ce qui mit fin à une des pages les plus noires de l'histoire de la Chine. Quel est le rapport avec les « design patterns » ? Le fait que pour beaucoup de ses premiers lecteurs le livre du GOF parut rédigé en chinois, et que, par la suite, il devint pour ces mêmes lecteurs leur petit livre rouge.

Ce livre peut être décrit comme un catalogue de vingt-trois solutions à des problèmes récurrents dans les applications informatiques. Dans le respect de l'esprit OO (ré-utilisabilité, adaptabilité, modularité), ces solutions ou design patterns sont décrites de manière suffisamment abstraite pour laisser une grande marge de manœuvre quant à leur implémentation finale. Ces design patterns sont répartis en trois catégories : « création », « structure » et « comportement ». Nous allons rapidement les décrire à l'attention des lecteurs désireux de ne pas aller plus loin dans le détail.

Parmi les patterns de « création », le plus connu et le plus mignon est sans nul doute le pattern « singleton », qui permet à une classe de ne produire qu'un et un seul objet. Vous pouvez tenter de découvrir la solution (un petit coup de pouce : un constructeur « privé » nous semble indispensable). Toujours parmi les « créations », le « factory method » et l'« abstract factory » sont une aide précieuse quand il s'agit de concevoir une classe ayant la responsabilité de créer des objets d'une autre classe. Parmi les patterns « structure », on trouve le pattern « composite », qui s'applique quand une classe devient un agrégat d'éléments simples ou de composites. Il est utilisé notamment dans le design des interfaces graphiques en Java. Le pattern « bridge » sépare l'interface de la classe de son implémentation, séparation que nous avons déjà largement évoquée et qui est une excellente pratique de conception OO. Le pattern « façade » regroupe un ensemble de classes dans une interface commune.

Le pattern « proxy » est particulièrement intéressant : il est indispensable à la conception d'applications distribuées, objet du chapitre 16. Le « proxy » a la même interface que l'objet distant, ce qui permet aux objets locaux de converser avec lui comme si l'objet n'était plus distant. Le proxy est alors responsable de l'emballage des messages et de la transmission de ceux-ci à l'objet réel, où qu'il se trouve. Dans la dernière catégorie, « comportement », on trouve le pattern « observer », auquel est entièrement consacré le chapitre 18 .

On trouve aussi le pattern « visitor », qui explique comment une opération itérative peut être effectuée sur un ensemble d'objets, sans que ceux-ci aient besoin de savoir qui est responsable de cette opération. Le pattern « memento » permet de prendre en compte des processus « transactionnels », car il donne de façon simple toute latitude pour revenir à l'état précédent d'un objet si, lors d'une série d'opérations qui doivent nécessairement se dérouler de la première à la dernière, cette série s'interrompt. Que du classique, donc !

En matière de design patterns, nos références françaises sont les suivantes :

- La bible : *Design patterns. Catalogue des modèles de conception réutilisables* » d'Éric Gamma, Richard Helm, Ralph Johnson, John Vlissides, chez Vuibert (il s'agit plutôt d'une traduction française de la bible).
- *UML et les design patterns,* de Craig Larman : bon ouvrage, très complet en ce qui concerne les mécanismes OO, le langage UML et la méthodologie RUP, mais un peu confus quant aux design patterns, quelque peu noyés dans une masse d'informations. Une étude de cas intéressante, néanmoins.
- *Design pattern par la pratique*, de Alan Shalloway et James Trott chez Eyrolles : plus introductif, plus accessible et aussi plus ciblé.
- *Design Patterns – Tête la première*, de Bert Bates, Eric Freeman, Elisabeth Freeman et Kathy Sierra chez O'Reilly : un excellent ouvrage, didactique à souhait, drôle et abondamment illustré.

Le fameux livre du GOF contient vingt-trois design patterns et, ô coïncidence, c'est le vingt-troisième chapitre du présent ouvrage qui vous les présente. Depuis, de nombreux livres en ont décrit de nouveaux, parfois dédiés à tel ou tel langage de programmation, mais c'est toujours le livre initial qui fait référence et trône dans la plupart des bibliothèques des professeurs d'OO. Dans ce livre, chaque pattern est présenté par son nom, associé au problème qui motive son recours, assorti de la solution proposée ainsi que des limites de sa mise en œuvre.

Toujours dans ce même livre, les patterns sont organisés en trois catégories :

- « créationnels », c'est-à-dire plutôt centrés sur la problématique de la construction d'objets grâce à l'amplification ou au court-circuit du constructeur ;
- « structurels », c'est-à-dire plutôt ciblés sur les schémas associatifs des classes, afin, par exemple, de contrebalancer l'héritage et l'association ;
- « comportementaux », c'est-à-dire décrivant plutôt des mécanismes astucieux à mettre en œuvre pour l'exécution des codes.

À cette catégorisation assez floue et arbitraire, nous préférerons par la suite une répartition plus souple, selon deux axes beaucoup plus simples à appréhender. Nous nommerons le premier « truc et ficelle » ; il regroupe, par exemple, les patterns singleton, « adapteur », « flyweight » et d'autres que nous allons découvrir tout de suite. En revanche, le deuxième axe respecte les principes de base de la programmation OO : modularisation, division du travail, encapsulation, stabilité, extensibilité… Enfin, tant que faire se peut, nous décrirons chacun de ces patterns dans le contexte des chapitres 20 et 22 (flipper, chimie et immunologie).

Les patterns « truc et ficelle »

Dans les années 1980, une série télé a popularisé un bricoleur de génie qui réussissait à se tirer de toutes les situations difficiles avec les seuls moyens du bord : un bout de bâton ou de ficelle, deux feuilles de papier ou d'arbre, ses lacets, son chewing-gum ou l'élastique de son caleçon. Ce héros de la débrouille s'appelait Mc Gyver et il a sans conteste fait beaucoup d'émules dans la communauté informatique, à commencer par les membres du GOF, qui se sont également employés à découvrir des solutions logicielles subtiles et efficaces dans des situations délicates. Alors, autant connaître ces solutions pour, le temps d'un chapitre, vous glisser dans la peau d'un Mc Gyver du petit écran, d'ordinateur, cette fois. Trois dernières précisions avant de nous lancer :

1. Lorsque nous illustrerons les patterns par du code, nous nous limiterons à la version Java de ces codes. Tout ce qui précède dans cet ouvrage devrait aisément, lorsqu'elles sont possibles, vous permettre d'en déduire les versions équivalentes dans les autres langages.
2. Chaque pattern pourrait donner lieu à des développements sans fin, mais nous nous bornerons ici à ébaucher les éléments essentiels qui les caractérisent, en laissant à d'autres ouvrages plus spécialisés le soin d'affiner et de prolonger cette description.
3. Nous passerons sous silence certains des vingt-trois patterns d'origine, afin de ne pas faire de ce dernier chapitre le plus indigeste de l'ouvrage. Notre choix est arbitraire, mais là encore, de multiples références vous permettront d'assouvir vos désirs de « patternité ».

Le pattern singleton

Dans la vie réelle, de nombreuses classes n'ont parfois comme raison d'être que la création d'un et un seul objet. Une première possibilité serait d'en faire des classes statiques, comme il en existe d'ailleurs pas mal dans la programmation OO. Dans un souci de cohérence (il est bien question de programmation orientée objet et non pas de programmation orientée classe), le GOF propose la mise au point d'une classe, dite classe « singleton », qui ne peut donner naissance qu'à une seule instance et garantir l'accès unique qu'à cette instance. Supposons que notre programme de flipper ne doive fonctionner qu'avec une et une seule boule. Le petit code Java qui suit vous permet de réaliser cela grâce à une classe `SingletonBoule`, qui ne permettra qu'à une seule boule de se balader dans notre flipper.

```
class SingletonBoule extends BouleDeFlipper {
 private static SingletonBoule b = null; // notez le static
 private static Object ob = new Object();

 private SingletonBoule(int x, int y, int r)  { // Le constructeur doit être privé
  super(x,y,r);
 }

 public static synchronized SingletonBoule getBoule(int x, int y, int r) {
   // notez le static et le synchronized
    synchronized (ob) {
    if (b == null) {
      b = new SingletonBoule(x, y, r);
    }
    return b;
  }
 }

… Continuation de la classe Boule

… Création de la boule unique

public void mousePressed(MouseEvent e) {
     int x = e.getX();
     int y = e.getY();

     if ((x > largeurDuFlipper - 40) && (y > hauteurDuFlipper - 40)) {
      laBoule = SingletonBoule.getBoule(x,y,15); // obtention de l'instance unique
```

```
 Thread nouveauThread = new FlipperThread (laBoule);
 nouveauThread.start();
}
```

Le constructeur est privé, ce qui rend impossible la création d'une boule en dehors de la classe. La méthode `getBoule()`, statique, vérifie si une instance est déjà créée. Si oui, elle la renvoie ; si non, elle la crée. Une complication supplémentaire apparaît dans le cas des applications multithread : en effet, si cette méthode s'interrompt après le test, mais avant la création de la boule, plusieurs boules pourraient être créées, jusqu'à une par thread, ce qui n'est évidemment pas souhaitable. Le rajout dans le code d'un mécanisme de synchronisation permet de contourner cette difficulté. La création de la boule suivra forcément, sans interruption possible, le résultat du test.

Le pattern adaptateur

Dans le code original du flipper, nous utilisons l'interface `ObstacleDuFlipper` dont une des méthodes abstraites `dessineToi(Graphics g)`, à redéfinir dans les classes précises d'obstacle, est responsable de l'apparition et de l'apparence de cet obstacle. Supposons qu'un ami développeur qui nous veut du bien propose de nous fournir pour l'obstacle `Champignon` une classe de sa fabrication, dénommée `ChampignonJoli`, dont l'énorme avantage est de proposer une méthode sophistiquée pour dessiner magnifiquement ce type d'obstacle et nommée `public void apparaîtBeauChampignon (Graphics g, Rectangle r)`. Grâce aux deux diagrammes UML de la figure 23-1 – l'un misant sur l'héritage et l'autre sur la composition – vous comprendrez aisément comment, sans modifier d'aucune manière le code d'origine, celui-ci pourra exploiter cette méthode. La manière dont la méthode `dessineToi` doit être redéfinie est indiquée pour les deux versions dans une note accolée à la classe `Champignon`. La classe qui sert d'intermédiaire entre l'interface de départ – celle que le flipper est contraint et forcé d'utiliser – et la classe contenant la méthode que nous souhaitons récupérer est dite `Adaptateur`. Il s'agit ici de la classe `Champignon`. Cet exemple montre comment, dans de nombreux cas, l'héritage et la composition peuvent apparaître comme une alternative pour un même problème.

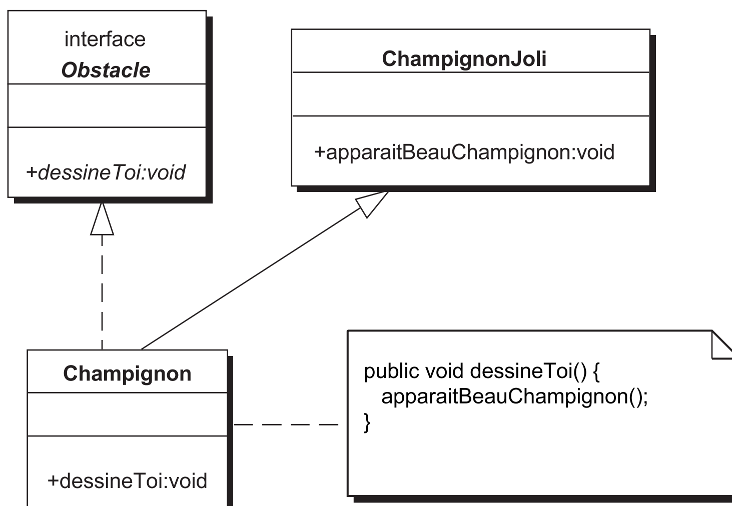
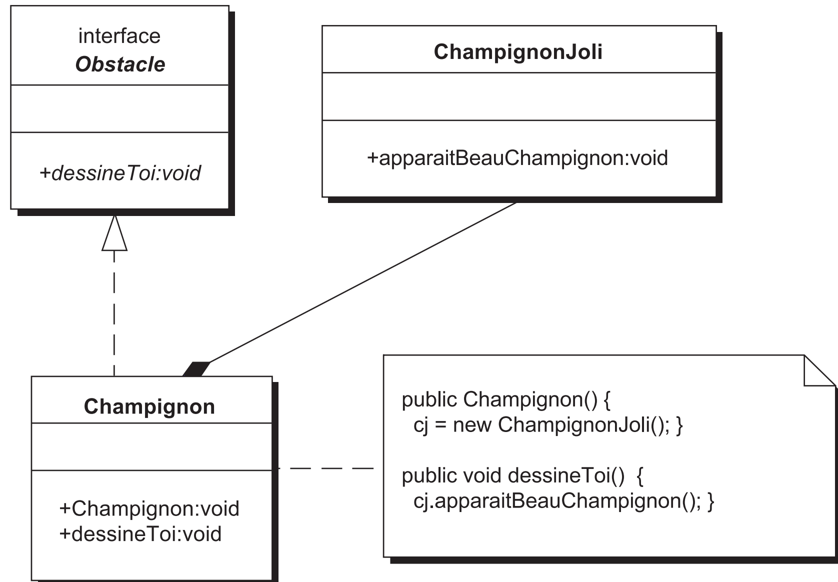


Figure 23-1

Le pattern « adaptateur » par héritage ou par composition.



Les patterns patron de méthode, proxy, observer et memento

Nous allons maintenant rapidement présenter quatre patterns pour le prix d'un, pour la bonne raison que nous les avons déjà rencontrés dans les chapitres précédents et qu'ils sont largement exploités dans les librairies Java. L'idée du pattern « patron de méthode » est de fournir un squelette du code dans lequel, bien qu'utilisée, une méthode agit sans son code. Il suffit alors au programmeur désirant exploiter ce squelette d'implémenter le corps de la méthode, par exemple, par la redéfinition d'une méthode provenant d'une interface, afin de bénéficier de son squelette. Celui-ci sera complété et adapté à ce corps. Il se caractérise donc par une partie fixe, proposée par un premier développeur (dans les librairies Java, par exemple) et par une partie variable (que fixe un deuxième développeur). Cette seconde partie se résume au corps de la méthode à définir. Elle est variable car elle dépend de la définition du corps.

Si, dans le flipper, les obstacles sont installés dans un vecteur, et qu'il nous vient l'envie de les réordonner dans ce même vecteur, la seule information qui fait défaut est le critère selon lequel les obstacles doivent se comparer et se réordonner. Toute la pratique de comparaison peut être mise en œuvre, à l'exception de ce critère de comparaison qui constitue ici, en fait, le corps manquant de la méthode, et qui est réintroduit comme indiqué dans le petit code qui suit :

```
public void actionPerformed(ActionEvent evt)  {
   Comparator c = new Comparator() {
      public int compare(Object o1, Object o2)
      {
        int t1 = (int)(((ObstacleDuFlipper)o1).renvoieMaZone().getWidth());
        int t2 = (int)(((ObstacleDuFlipper)o2).renvoieMaZone().getWidth());
        return (t1 - t2);
      }
    };
```

```
      Collections.sort(lesObstacles,c);
      Iterator i = lesObstacles.iterator();
      while (i.hasNext())
      {
         System.out.println(i.next());
      }
   }
```

La définition du critère de comparaison se fait par l'intermédiaire du corps de la méthode `compare` prévue dans l'interface `Comparator` et censée renvoyée un entier (– 1, 0 ou 1 suivant le résultat de la comparaison). Ici, deux obstacles sont comparés sur la base de leur largeur. Il suffit alors d'appeler sur la classe `Collections`, la méthode `sort` déjà écrite dans les librairies Java (il existe de multiples manières de trier un vecteur, de nombreuses sont très mauvaises en termes de performance. Le `sort` déjà programmé dans la librairie Java est optimal), de lui passer la classe `c` implémentant l'interface `Comparator` et le tour est joué ! Cette manière de faire est particulière au pattern patron de méthode. Elle est utile chaque fois qu'une large partie d'un code est fixe et connue et que quelques adaptations doivent être fournies par la redéfinition d'une méthode.

Nous avons rencontré le pattern proxy dans le chapitre consacré aux objets distribués. La présence du stub côté client, appelé à jouer le rôle du serveur, en est l'illustration la plus connue. Le proxy se substitue à un objet manquant ou distant, afin de ne pas modifier radicalement le code de son interlocuteur, tout en apportant quelques fonctionnalités additionnelles indispensables au fonctionnement de l'objet distant. Un proxy serait le bienvenu si, parmi les obstacles du flipper, l'un d'entre eux prend énormément de temps à se dessiner car il doit rechercher sur le Web une image particulière. Comme indiqué dans la figure 23-2, et de manière très semblable au pattern « adaptateur », il est possible de remplacer cet obstacle par un proxy qui, tout en déclenchant la méthode appropriée sur l'obstacle de départ, ajoute quelques fonctionnalités qui permettent de patienter, par exemple un message signalant que la recherche ou le téléchargement sont en cours.

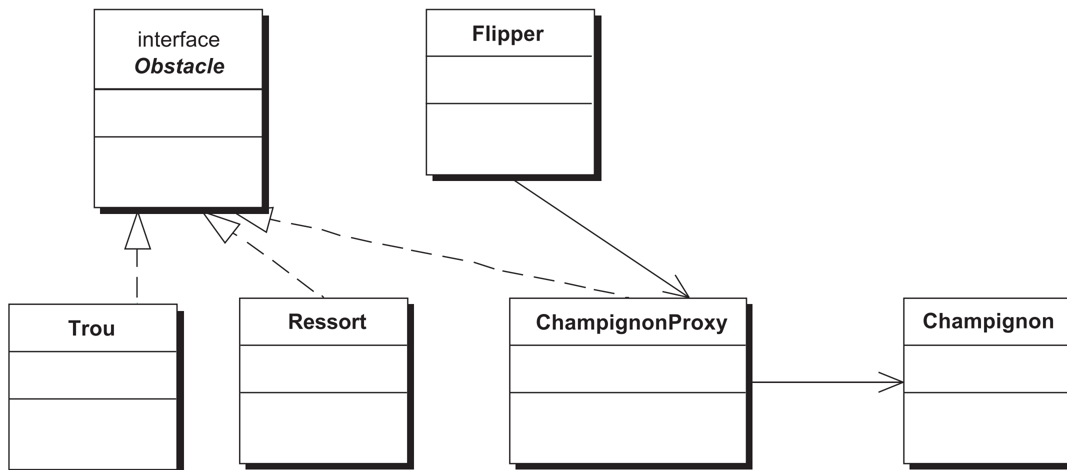


Figure 23-2

La classe « proxy » sert d'intermédiare entre le flipper et le champignon.

Le chapitre 18 est dédié au pattern `observer` dont le but est de maintenir deux objets synchronisés bien qu'ils ne possèdent pas entre eux de lien explicite. L'implantation proposée par Java dans ses librairies – qui découle de ce pattern – fait appel à une classe `Observable` et à une interface `Observer`.

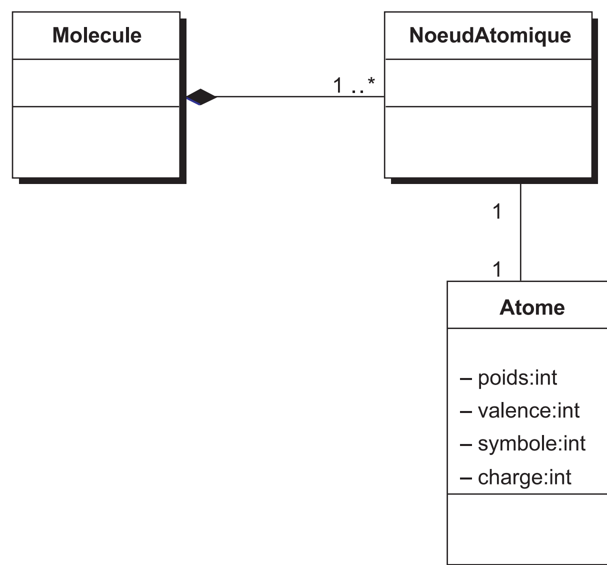
Enfin, le pattern `memento` vous permet de revenir à la case départ si un ensemble de manipulations d'objet n'aboutissent pas et qu'il devient important de restaurer les objets modifiés dans leur état d'origine. Il suffit soit de les stocker au départ des manipulations sur le disque dur par une des méthodes décrites dans le chapitre 19, soit de les cloner et de conserver ces copies pendant toute la durée des manipulations. En général, la classe de l'objet à mémoriser intègre un mécanisme de génération d'un « memento », qui ne conserve que les attributs susceptibles de modification et qu'il est important de pouvoir restituer.

Le pattern flyweight

Dans le chapitre précédent, consacré à la modélisation et à l'implémentation d'un réacteur chimique, on trouve la classe `Molecule` qui, comme chacun sait, est un réseau d'atomes. Chaque atome possède des informations qui lui sont propres, reprises dans la classe : son symbole, son poids atomique, sa valence, sa charge éventuelle, etc. Une première solution rapide pour réaliser la classe `Molecule` aurait pu se limiter à une relation de composition entre la classe `Molecule` et la classe `Atome`. Cependant, lorsqu'on sait qu'une même molécule organique peut contenir jusqu'à des millions d'instances du même atome de carbone, il est stupide, et surtout extrêmement coûteux en mémoire, de répliquer pour chacune de ces instances toutes les informations comme le poids, la valence ou le symbole. Le remède à cela, application directe du pattern `flyweight` dont le but essentiel est de réduire l'espace de stockage des objets, consiste, comme indiqué dans la figure 23-3, à regrouper toutes ces informations dans une classe à part. L'économie de mémoire peut très vite s'avérer considérable. Nous retrouvons ce même besoin de stockage économe lorsque les cellules du système immunitaire se clonent. Il n'est sans doute pas nécessaire de stocker dans chaque clone l'entièreté de l'information. Ainsi, le même patrimoine génétique se retrouve pour l'essentiel dans toutes les cellules, ce qui autorise à ne le stocker qu'une fois en mémoire, toutes les cellules pouvant alors y faire référence.

Figure 23-3

Principe du pattern « flyweight »



Les patterns builder et prototype

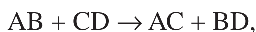
Puisque nous sommes dans la chimie et l'immunologie, restons-y. Un type d'objet dont la construction s'avère délicate dans le réacteur chimique sont les molécules. Le programme du réacteur démarre avec un petit nombre de molécules. De nouvelles molécules apparaissent et disparaissent au cours de la simulation, résultant des réactions impliquant celles s'y trouvant au départ. En général, le programme lit les molécules de départ à partir d'un fichier ou d'une fenêtre initiale, dans lesquels celles-ci sont décrites à l'aide de leur symbole : CH_4 ou NH_3. Il faut alors pouvoir transcrire ces formules en un graphe canonisé, car c'est sous cette forme qu'elles seront ensuite traitées dans le code. Cette transformation de l'expression symbolique au graphe ne se fait pas sans peine, car elle exige une lecture et un traitement attentifs des expressions définissant les molécules initiales.

Comme indiqué dans le chapitre précédent, lorsqu'une réaction se produit, de nouvelles molécules apparaissent. Là encore, la génération d'une nouvelle molécule ne se fait pas sans mal et exige de nombreuses manipulations des molécules de départ.

La solution qui vient immédiatement à l'esprit consiste à prévoir et à installer tous ces traitements dans le ou les constructeurs de la classe Molécule. La solution préconisée par le GOF est tout autre. Dès que la genèse de nouveaux objets exige une séquence de traitements compliqués (par exemple : lecture et parsing d'une chaîne de caractères), il propose de séparer cette séquence de la construction de la molécule à proprement parler et de l'installer dans une classe à part, ici la classe `MolecularBuilder` (voir le code ci-dessous), dont une ou plusieurs méthodes renverront une nouvelle instance de la classe `Molecule`, aboutissement de ces nombreux traitements. Le constructeur de la classe `Molecule` se limite à une création très élémentaire de l'objet, précédant ou concluant tous ces traitements.

```
public class MolecularBuilder
{
  ……
  public static Molecule build(String s) throws BuilderException
  {
    Molecule m = new Molecule();
    ……
    // de nombreuses instructions de manipulation de « m » afin de construire la nouvelle molécule
      }
    }
    return m; // renvoie la nouvelle molécule
  }
}

… Et quelque part dans le main, on crée une nouvelle molécule :

try {
         m = MolecularBuilder.build(s);
      }
 catch (BuilderException e) {
         JOptionPane.showMessageDialog(null, "rentrez " +
                         "une nouvelle Molécule");
         OK = false;
     }
```

Lorsqu'une nouvelle molécule apparaît à l'issue d'une réaction comme celle-ci :



$$AB + CD \rightarrow AC + BD,$$

cette nouvelle venue ne se construit pas à partir de rien, mais récupère les atomes des molécules de départ, avant la réaction. Ainsi, dans la réaction qui précède, la molécule AC est en partie une combinaison des molécules AB et CD. La manière la plus simple de générer le nouvel objet moléculaire est donc de cloner en partie les deux molécules de départ et de construire la nouvelle venue à partir de ces deux clones. Cette démarche nous conduit tout droit au pattern `prototype` dont c'est la raison d'être : construire un nouvel objet à partir du clone d'un objet existant. En Java, et comme nous l'avons vu en détail dans le chapitre 14, la méthode « clone » de la classe `Object` est là, prête à l'emploi, pour assurer ce clonage. On se rappelle que la méthode clone est définie comme protected, car elle requiert souvent, dans le cas de copies en profondeur, une redéfinition faisant appel à la version de la classe `Object`. Lorsqu'on clone une molécule, c'est en fait son graphe qu'on clone, en parcourant tous les nœuds de celui-ci. Le clonage d'un nœud atomique est donc défini récursivement et entraîne le clonage de tous les nœuds atomiques avec lesquels ce premier nœud est connecté.

De même, comme toute cellule biologique, chaque cellule immunitaire T (`TCell`) est capable de se dupliquer plusieurs fois. Le code Java qui s'occupe du clonage de la cellule est en partie repris ci-dessous ;

```
public Object clone()
      {
           try
           {
                 TCell obj = (TCell)super.clone();

                 return obj;
           }
           catch(CloneNotSupportedException e)
           {
                 throw new InternalError();
           }
      }
```

Le pattern façade

Comme parfaitement illustré par les diagrammes de classe et de séquence de la figure 23-4, ce pattern dissimule un ensemble d'objets sous une interface unique, la seule avec laquelle l'utilisateur de tous ces objets interagira. C'est cette interface unifiée qui joue le rôle de façade. Elle permet une interaction simplifiée avec cet ensemble d'objets ainsi qu'un point de contact unique. L'interlocuteur n'a nullement besoin de connaître la façon dont tous les objets agissent pour satisfaire les services proposés par l'interface. C'est la version logicielle de l'arbre qui cache la forêt.

Les patterns qui se jettent à l'OO

Les sept derniers patterns que nous allons présenter, deux pour le flipper et cinq pour la chimie, sont légèrement plus subtils que ceux vus jusqu'à présent, mais surtout, ils s'inscrivent parfaitement dans les principes de base de la programmation OO. Ces patterns sont des révélateurs de la véritable nature de l'OO et leur maîtrise garantit une bonne compréhension et une compétence réelle dans la pratique de l'OO. Parmi ces principes, il en est un majeur, déjà vu dans les chapitres précédents mais sur lequel il est important de revenir et réinsister. La programmation OO tente de sauvegarder au maximum de larges espaces de variation ou d'extension de parties de code sans que cela ait d'impact sur le reste de celui-ci.

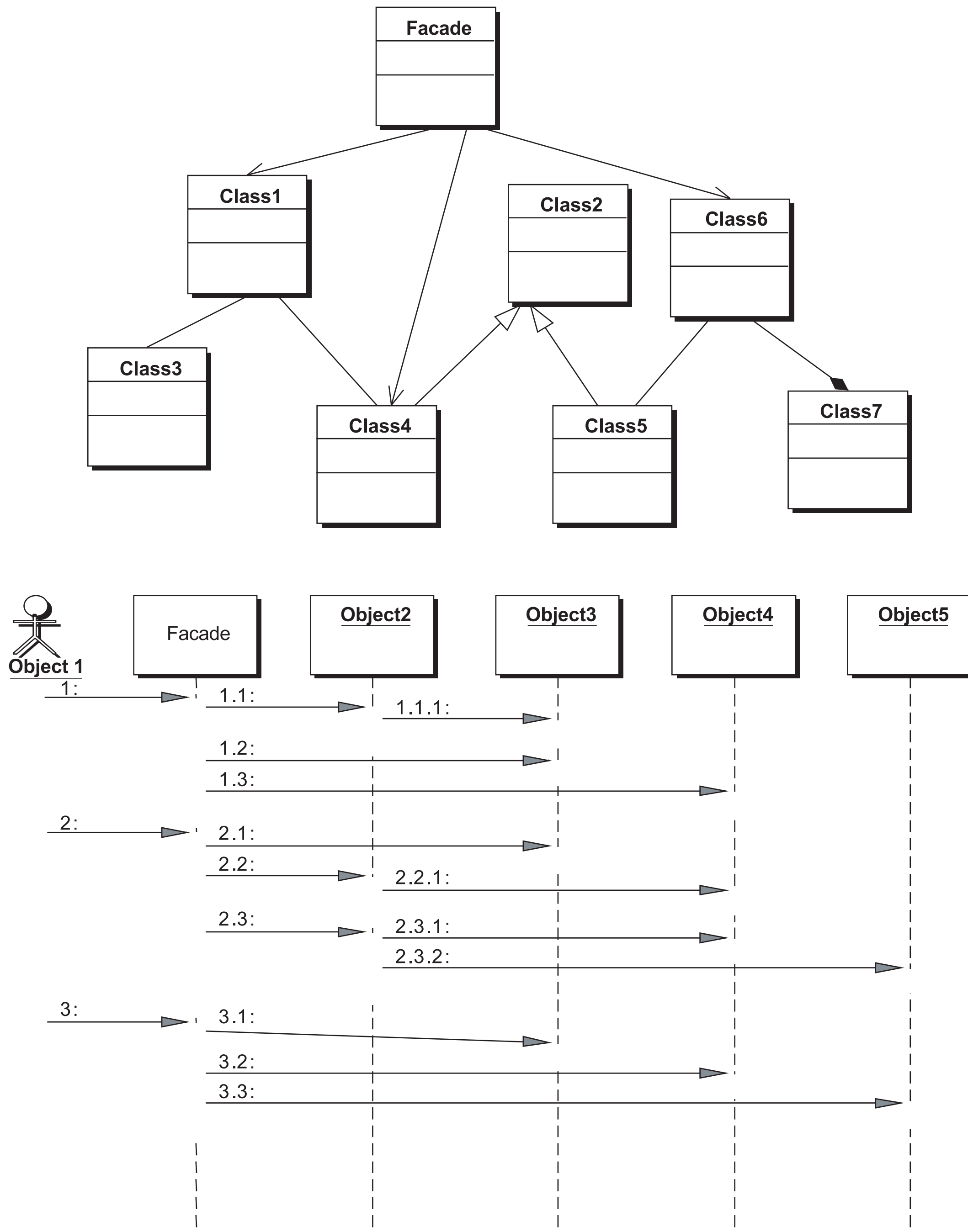


Figure 23-4

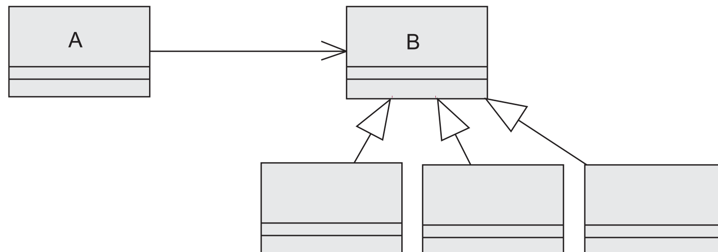
Principe du pattern « façade ».

En substance, tous les design patterns qui suivent ont pour motivation essentielle la mise au point de codes modulaires, facilement modifiables et extensibles, tout en maintenant une très grande stabilité.

Nous savons que le mécanisme d'encapsulation, découlant de l'emploi des mots-clés `private` ou `protected` permet un premier niveau de stabilité. Ce qui est « private », attribut comme méthode, peut être aisément modifié dans une classe sans que cela n'exige en rien de modifier les autres classes. La modularisation en soi permet également d'accroître la stabilité des codes, en répartissant le plus et le mieux possible les responsabilités entre les différentes classes (les classes `adaptateur`, `proxy` ou `façade`, présentées ci-dessus, participent déjà à cela). Mais c'est par l'entremise de l'héritage et du polymorphisme, que l'encapsulation est portée au zénith, car une classe peut apparaître pour le compilateur comme interagissant avec la superclasse de plusieurs autres alors qu'à l'exécution, ce sont bien ces dernières qui détermineront le comportement des objets. Ainsi, lorsqu'un objet de la classe A de la figure 23-5 interagit avec plusieurs objets de la superclasse B, tout ce qui se trouve en dessous de B dans le diagramme est hors du « champ de vision » (et donc de compilation) de A et, en conséquence, directement modifiable sans que A ne s'en trouve affecté.

Il est possible de modifier, d'ajouter ou de supprimer des sous-classes de B sans que cela ne perturbe en rien le code de A. Mais commençons par deux premiers patterns tout à fait délicieux, reposant sur la composition, l'héritage et le polymorphisme, et appliqués tout deux au flipper.

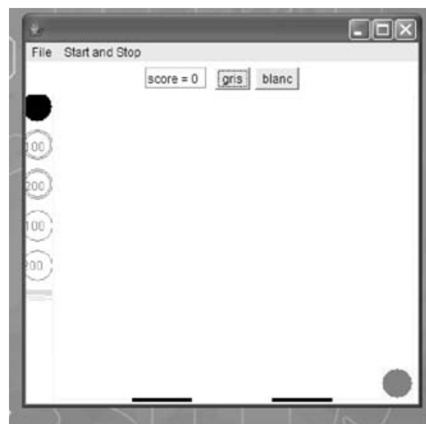
Figure 23-5
Une forme supérieure d'encapsulation.



Le pattern command

Supposons que l'on souhaite enrichir l'interface du flipper avec quelques menus et quelques boutons comme le montre la figure 23.6.

Figure 23-6
Nouvelle interface du flipper.



Chaque « item » du menu et chaque bouton se voit associer une fonctionnalité donnée, comme c'est le cas dans le code Java classique qui suit :

```
public void actionPerformed(ActionEvent evt) {
   Object obj = evt.getSource();
   if (obj == mOpen) { // item de menu "mOpen"
     System.out.println("Open");
   }
   if (obj == mSave) { //item de menu "mSave"
     System.out.println("Save");
   }
   if (obj == mStarting) {
          System.out.println("Start");
   }
   if (obj == mStopping) {
     System.out.println("Stop");
   }
   if (obj == unBoutonGris) { // le bouton gris
     System.out.println("Gray");
     setBackground(Color.GRAY);
   }
   if (obj == unBoutonBlanc) { // le bouton blanc
     System.out.println("Blanc");
     setBackground(Color.WHITE);
   }
```

Les boutons et les menus auront été préalablement créés et associés à une interface `ActionListener` comme suit pour le seul bouton blanc :

```
unBoutonBlanc = new Button("blanc");
unBoutonBlanc.addActionListener(this);
```

Le corps de la méthode `actionPerformed` reprend toutes les fonctionnalités associées à ces éléments de l'interface. Cette façon de procéder est loin d'être satisfaisante, et ce pour deux raisons principales. D'abord la présence de cette séquence de test (on aurait pu tout aussi bien utiliser un « switch » à la place) dont le principal défaut est le manque de stabilité si l'idée nous vient d'ajouter quoi que ce soit dans l'interface. Dès qu'une suite de tests conditionnels ou un switch apparaît, il est toujours bon de vous demander, à la condition que vous soyez devenu un vrai mordu de l'OO, bien sûr (mais à ce stade du livre le contraire serait malheureux), s'il n'y aurait pas lieu de remplacer le tout par une structure d'héritage. Si c'est le cas, vous accroîtrez en tout état de cause la stabilité du code face à des ajouts ou à des retraits des conditions antérieures dans le test.

La deuxième raison est que la méthode `actionPerformed()` est encombrée avec toutes les fonctionnalités associées à chacun des éléments de l'interface, ce qui la rend lourde et multiforme, c'est-à-dire peu en phase avec la simplicité et la modularité derrière lesquelles courent les développeurs OO.

L'alternative est l'application du pattern `command`, dans lequel chaque tâche de l'interface se définit dans une classe à part qui implémente une interface commune contenant la méthode abstraite et polymorphe `execute()`. Plus de problème de stabilité et modularité assurée, comme le code Java qui suit et qui illustre cette méthode le démontre de façon convaincante. Nous limitons ce code à la fonctionnalité d'un des boutons et d'un des menus.

```
// Tout d'abord définition des fonctionnalités de chaque élément de l'interface dans une classe à part
// implémentant l'interface Command

public class BoutonGrisCommand extends Button implements Command {
 Frame f;

 public BoutonGrisCommand(String label, Frame f) {
  super (label);
  this.f = f;
 }

 public void execute() { // C'est la méthode polymorphe qui reprend les fonctionnalités
  f.setBackground(Color.GRAY);
 }
}


public class FileOpenCommandextends MenuItem implements Command (
 public FileOpenCommand(String label){
  super(label);
 }

 public void execute(){
  System.out.println("Open");
 }
}

……………………………
// Création des nouveaux objets boutons et menus à partir de ces classes

mOpen = new FileOpenCommand("Open");
unBoutonGris = new BoutonGrisCommand("gris", this);

……………………………
// Ajout de l'interface ActionListener, comme classiquement fait

mOpen.addActionListener(this);
unBoutonGris.addActionListener(this);

// Nouvelle méthode « actionPerformed » … C'est là que ce pattern prend tout sa signification
// Les tests ont disparu et la méthode est beaucoup plus légère

public void actionPerformed(ActionEvent evt) {
  Object obj = evt.getSource();
  if (obj instanceof Command)
  {
      ((Command)obj).execute();
  }
 }
}
```

Le pattern décorateur

Dans l'état actuel du programme, le flipper contient un ensemble prédéfini d'obstacles : champignon, champignon à points, ressort, trou, etc. Ces catégories sont fixées une fois pour toute et se singularisent par l'effet de l'obstacle sur la balle. Or imaginons que nous désirons, lors de l'installation des obstacles dans le flipper, apporter plus de souplesse au comportement de ceux-ci. Alors qu'il est obligatoire que tous les obstacles de type champignon fassent rebondir la balle, on souhaiterait que certains, en plus, incrémentent le score, que d'autres changent de couleur, que d'autres encore puissent faire les deux, un troisième comportement étant même envisageable, à la demande : que l'obstacle champignon émette une petite musique au moment du choc. En gros, on souhaite que différents objets champignons puissent se caractériser, à la carte, par un ensemble de comportements différents, tous issus d'un nombre fini de possibilités de base.

Une première solution extrêmement lourde et naïve est d'imaginer autant de classes (et donc toutes les combinaisons d'héritage qui le permettent) qu'il y aurait de types d'obstacles différents. Une classe serait de type « Point-Couleur », une autre « Couleur-Son », une troisième « Point-Son-Couleur », etc. On pressent aisément la prolifération de toutes ces classes avec l'accroissement du nombre de fonctionnalités de base possibles. Une solution bien plus élégante est fournie par le pattern « décorateur », dont le diagramme de classe (figure 23-7) et le code Java correspondant suivent.

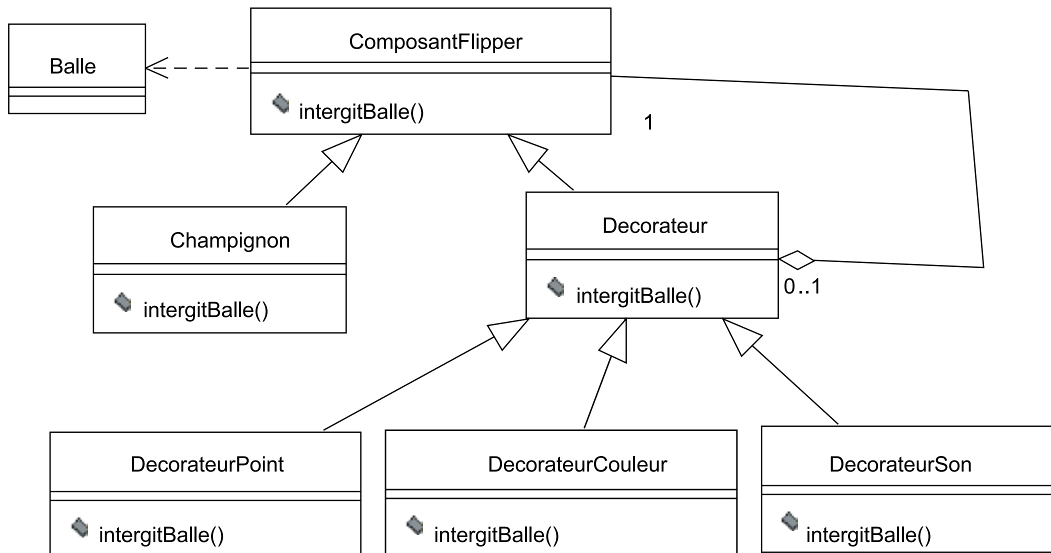


Figure 23-7

Le pattern « décorateur ».

```
import java.awt.*;

abstract class ComposantFlipper {
 public abstract void interagitBalle();
}

class Champignon extends ComposantFlipper {
```

```
 public void interagitBalle() {
          // Fait juste rebondir la balle
 }
}

abstract class Decorateur extends ComposantFlipper {
  private ComposantFlipper unComp;

  public Decorateur (ComposantFlipper unComp) {
          this.unComp = unComp;
  }

  public void interagitBalle() {
         if (unComp != null) unComp.interagitBalle();
  }
}

class DecorateurPoint extends Decorateur {
  public DecorateurPoint(ComposantFlipper unComp) {
         super (unComp);
  }

  public void interagitBalle() {
      // Incrémente les points
      super.interagitBalle();
  }
}

class DecorateurCouleur extends Decrateurs {
  public DecorateurCouleur(ComposantFlipper unComp) {
         super (unComp);
  }

  public  void interagitBalle() {
      // Change de couleur…;
      super.interagitBalle();
  }
}
class DecorateurSon extends Decorateur {
  public DecorateurSon(ComposantFlipper unComp) {
         super (unComp);
  }

  public void interagitBalle() {
      // Fait une petite musique…;
      super.interagitBalle();
  }
}

public class Flipper {
  public static void main(String args) {
```

```
        // Lors de la création de l'obstacle vous ajoutez à la carte,
        // de manière très souple,toutees les fonction désirées

           ComposantFlipper c =
  new DecorateurSon(new DecorateurPoint  (new  DecorateurCouleur (new Champignon())));
           c.interagitBalle();
    }
  }
```

Le pattern décorateur permet d'enchaîner de manière très souple une succession de fonctionnalités à un même objet, en court-circuitant pour ce faire toutes les combinaisons possibles d'héritage. C'est ce même décorateur qui est à l'œuvre dans Java lors de l'utilisation des classes Stream (chapitre 19), et que vous combinez de manière flexible des objets issus des sous-classes de Stream (par exemple `new BufferedInputStream (new InputStream)`). Tous ces décorateurs sont dérivés de la classe FilterInputStream. Il est intéressant de constater qu'un décorateur agrège et hérite à la fois d'un composant, comme un `BufferedInputStream`, qui hérite d'un `InputStream` tout en pouvant en agréger un autre. En immunologie, les types cellulaires pourraient se distinguer par un ensemble de fonctionnalités de base qu'ils seraient susceptibles de présenter ou pas. Le diagramme de classe de la figure 23-8 illustre cette situation ainsi que la présence à nouveau du pattern décorateur mais cette fois appliqué à la biologie.

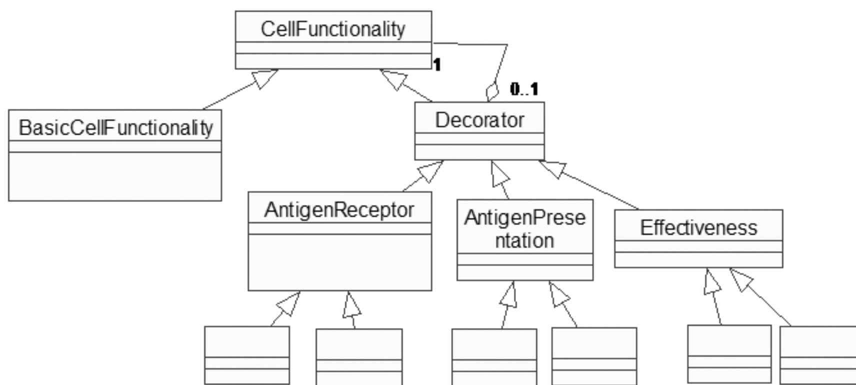


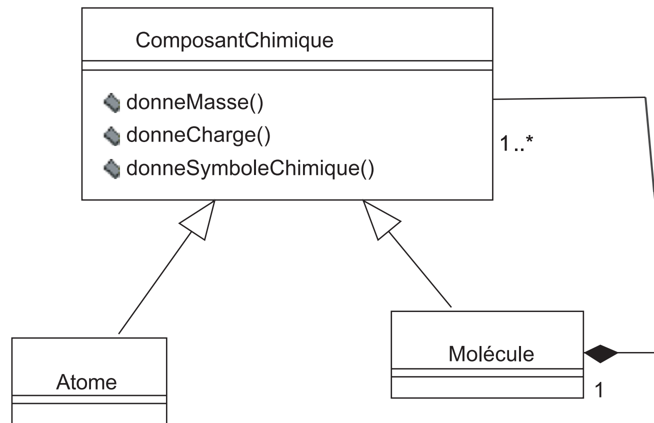
Figure 23-8
Le pattern décorateur appliqué à l'immunologie

Le pattern composite

En tordant quelque peu le cou à la chimie (les chimistes, en perdition et en quête d'une nouvelle identité, qui liront un jour ce livre nous le pardonneront), nous allons présenter le pattern composite. Celui-ci est idéal pour créer des structures complexes dans lesquelles, à l'instar des poupées russes, des éléments se trouvent imbriqués les uns dans les autres ou, comme dans un mille-feuille, empilés les uns sur les autres. De manière générale, un « composite » est un groupe d'objets dans lequel certains objets peuvent en contenir d'autres. Certains objets seront donc plutôt de type groupe (mais pouvant contenir d'autres groupes), comme nos molécules, alors que d'autres seront isolés et singuliers, tout comme nos atomes.

Non seulement un groupe contiendra soit un ensemble d'objets individués, soit d'autres groupes mais, plus important encore, tant les groupes que les objets individués présenteront partiellement un comportement commun. Le diagramme de classe de la figure 23-9 rend bien compte de ce pattern.

Figure 23-9
Le pattern « composite » appliqué à la chimie.

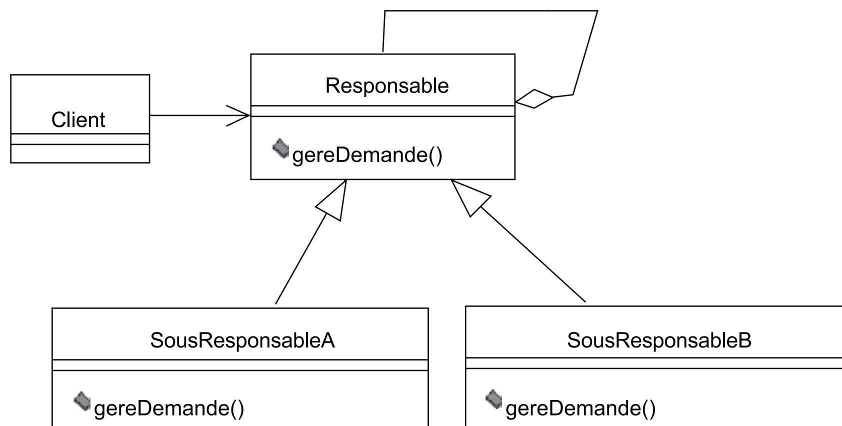


Dans ce diagramme, on considère (et c'est là que les chimistes écarquilleront les yeux à juste raison) que les molécules, en plus d'être un ensemble d'atomes, peuvent être considérées ici comme un ensemble de molécules. Les trois méthodes installées uniquement dans la superclasse doivent être redéfinies dans les deux sous-classes mais existent bel et bien dans ces deux-là dans des versions différentes. Tout comme un atome, une molécule a une masse, une charge et un symbole chimique. Un cas très parlant d'application de ce pattern est la conception d'interfaces graphiques en Java à l'aide des classes `Components` et `Container`. On le retrouve également à l'œuvre dans certaines structures arborescentes (comme le langage XML), dans lequel chaque nœud de l'arbre pointe vers un ensemble de nœuds jusqu'à atteindre les feuilles de l'arbre. Une classe additionnelle, la classe `Composant`, a pour rôle de factoriser les attributs et les méthodes communs aux nœuds et aux feuilles.

Le pattern chain of responsabililty

Ce pattern se comprend aisément en observant le diagramme de classe de la figure 23-10.

Figure 23-10
Le pattern « chain of responsibility ».



La superclasse `Responsable` a, entre autres responsabilités, celle de répondre à une demande adressée par le client. À son niveau, la méthode `gereDemande` est abstraite et se trouve redéfinie dans un ensemble de sous-classes qui, toutes, ont la possibilité soit de gérer cette demande soit, si elles en sont incapables, de déléguer cette gestion à une autre des sous-classes. C'est la raison pour laquelle la superclasse se réfère elle-même par un lien d'agrégation, afin de pouvoir « refiler la patate chaude » à une autre des sous-classes. Chaque sous-classe peut se refiler la demande, jusqu'à aboutir à celle d'entre elles dont la méthode `gereDemande` est capable de satisfaire le client.

Dans notre pattern chimique composite du chapitre précédent, si nous demandons à une molécule, composée d'autres molécules, de nous donner sa masse, elle ne peut qu'additionner la sienne à celles qu'elle demande à toutes celles dont elle est composée, comme dans le code qui suit :

```
public double donneMasse(){
   double masse = 0;
   for (int i=0; i<lesComposants.size(); i++)
   {
     masse += ((ComposantChimique)lesComposants.elementAt(i)).donneMasse();
   }
   return masse;
}
```

Cette illustration est quelque peu triviale car tous les composants seront capables, en partie, d'assumer leur responsabilité. Imaginez un nouvel atome errant, cherchant à s'installer quelque part dans un immense graphe moléculaire. Il peut tenter le coup avec la première molécule rencontrée qui, après vérification, l'accepte ou pas. Dans la négative, elle peut déléguer cette responsabilité aux autres molécules avec lesquelles elle se trouve connectée, jusqu'à en trouver une qui accepte d'héberger cette pauvre âme atomique en peine.

Les patterns strategy, state et bridge

Nous allons terminer notre passage en revue des design patterns par une description rapide des trois derniers. Nous les avons regroupés car, à nouveau, héritage et polymorphisme aidant, ils présentent de très fortes ressemblances. Le pattern strategy est une application directe du polymorphisme, très largement décrit dans le chapitre 13. Dans ce pattern, différentes sous-classes implémentent de manière différente une même fonctionnalité stratégique. Le client n'a pas à se préoccuper de connaître les différentes implémentations ; celles-ci seront finalement choisies au moment de l'exécution du code en fonction du type précis de l'objet interlocuteur implémentant la version de la stratégie souhaitée. Ainsi, dans la figure 23-11, lorsque le réacteur demande à une réaction de s'exécuter, cette réaction s'effectuera différemment et selon son mode propre, en fonction de sa nature définitive.

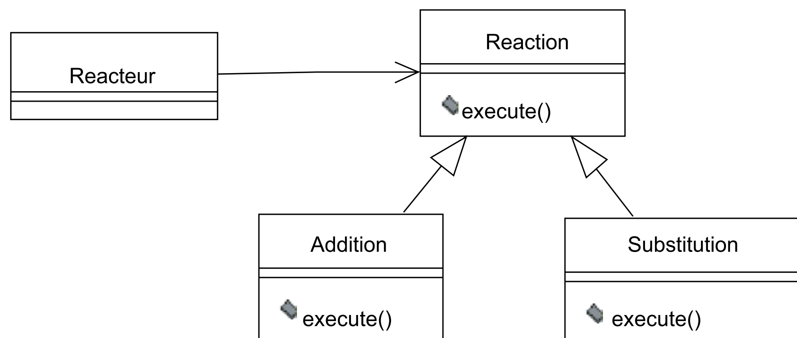


Figure 23-11
Le pattern « strategy ».

Comme les chimistes le savent bien, une molécule peut se trouver à un moment donné dans différents états, cet état variant au cours du temps : au repos, excitée, et si excitée, dans différents états possibles d'excitation. Selon l'état dans lequel elle se trouve, la molécule peut se comporter différemment, notamment dans ses réactions avec d'autres molécules. La manière dont les molécules réagissent et se comportent en général dépend donc de l'état dans lequel elles se trouvent à un moment donné. Créer autant de sous-classes de molécules qu'il y a d'états possibles n'est pas une solution au problème, puisqu'une molécule peut changer d'état au cours du temps alors qu'un objet ne peut changer de classe au cours d'une simulation. La solution proposée par le GOF, comme le montre le diagramme de classe de la figure 23-12, est d'associer la molécule à une classe « EtatMoleculaire » qui, elle, possèdera autant de sous-classes qu'il y a d'états possibles, et d'installer à même ces classes et sous-classes tout ce qui permet de modifier le comportement des molécules selon leur état. Il faut donc déléguer toutes les opérations qui dépendent de l'état de la molécule à son objet « état courant ». Les transitions entre états peuvent se produire dans une classe extérieure `Reacteur` ou se faire à même les classes d'état. Nous avons exploité ce pattern dans le chapitre précédent, pour programmer la transition des cellules immunitaires entre différents états : « naïf », « excité », « effectif » et « mémoire ». Ce pattern accompagne comme il se doit toute mise en œuvre d'un diagramme UML d'état-transition.

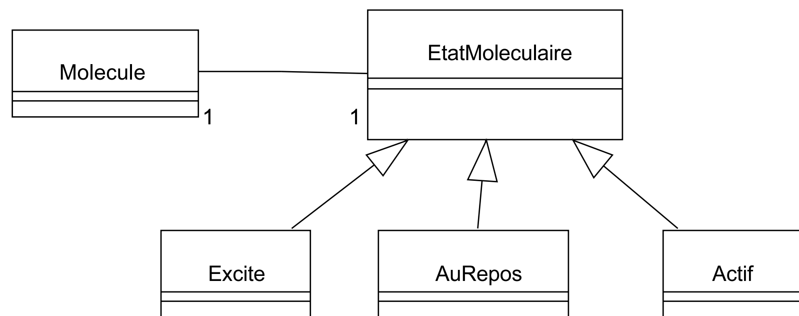


Figure 23-12
Illustration chimique du pattern « state ».

Une autre très parlante illustration de ce même pattern nous permettrait d'améliorer l'analyse de classe faite au chapitre 19 de notre application de réservation de spectacles. En effet, nous avons considéré que les clients de cette application pouvaient être de deux types : « abonné » ou « non abonné ». Selon leur statut, leur manière de réserver ou payer s'en trouvera différenciée. La solution adoptée fut l'héritage d'une super-classe `client` par deux sous-classes `abonné` et `non abonné`. Nous voyons aisément le défaut inhérent à cette solution : un client restera abonné ou non pour la vie et ne pourra plus changer de statut une fois créé. Or une solution bien plus flexible permet aux clients de changer d'état : celle d'appliquer encore une fois le pattern `state` comme illustré dans la figure 23-13. Les manières de réserver ou de payer qui se différencient selon le statut du client seront redéfinies dans les deux sous-classes d'état. Ainsi, par exemple, le client définira sa manière de réserver (en fait, il la délèguera à son état courant) comme suit :

```
class Client {
   Etat etatCourant ;
   public void réserver () {
       etatCourant.réserver() ;
   }
   public void changeEtat(Etat nouvelEtat) {
       etatCourant = nouvelEtat ;
   }
}
```

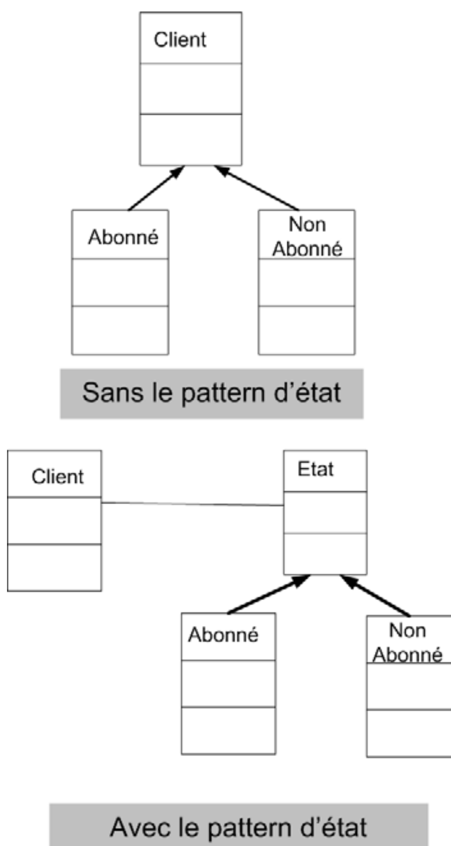


Figure 23-13
Autre illustration du pattern state.

Une dernière chose que les chimistes savent bien (et puis, c'est promis, on leur fiche définitivement la paix), c'est qu'en fonction de sa « radicalité » (non radical, mono ou bi-radical) ou éventuellement de l'intensité de sa charge électrique (–, – –, +, ++), un composant chimique, quel qu'il soit, atome ou molécule, peut se comporter très différemment. Une solution, naïve comme il se doit, serait d'abuser outrageusement de l'héritage et d'imaginer autant de classes qu'il n'y a de combinaisons possibles : « atome-monoradical », « atome-non radical », « atome-biradical », « molecule – ++ », etc. Si nous nous limitons à 2 types de composant chimique, 3 versions possibles de la radicalité et 4 valeurs possibles pour la charge électrique, cela nous fait au total, 2 ∞ 3 ∞ 4 = 24 classes différentes combinant toutes les possibilités. De plus, il suffit d'ajouter telle ou telle possibilité concernant soit la radicalité, soit la charge, pour que tout le diagramme de classe s'en trouve modifié. Une solution élégante à cette explosion et à cette instabilité potentielle est de recourir, comme le diagramme de classe de la figure 23.14 le montre, au pattern « pont ».

Cette solution, qui sépare clairement le type des composants chimiques de certaines de leurs propriétés fonctionnelles, permet de modifier la nature « radicalaire » ou « électrique » de ces composants sans apporter de grosses modifications au code, tant à la rédaction qu'à son exécution. On retrouve ce même souci en immunologie si l'on considère, comme la figure 23-14 le propose, que les cellules présentatrices d'antigène, quelles qu'elles soient, peuvent présenter sur leur surface n'importe quel type de molécule MHC. Que cela soit possible ou non, seul les immunologistes sont capables de nous répondre. Mais au moins, l'existence de ce diagramme

les emmenera à se poser la question. Nous pensons qu'un des avantages souvent sous-estimé des technologies OO, des langages de programmation, d'UML et des design patterns, est d'aider le développeur à mieux comprendre cette réalité qu'il voudrait pouvoir reproduire à même son code. Programmer force à désambiguïser tout ce que l'on sait de cette réalité que l'on souhaite transposer dans son programme. L'ordinateur ne peut se contenter d'un flou artistique dont pourtant la chimie ou la biologie ne se privent pas, confiantes qu'elles sont que leurs praticiens réussissent malgré tout à se comprendre. Par l'application du pattern pont, toutes les variations dans les options sont clairement maintenues à part, avec comme principal résultat de simplifier (pas d'explosion d'héritage) et de stabiliser le code, comme d'habitude. C'est sûr, à force, vous allez finir par connaître la musique.

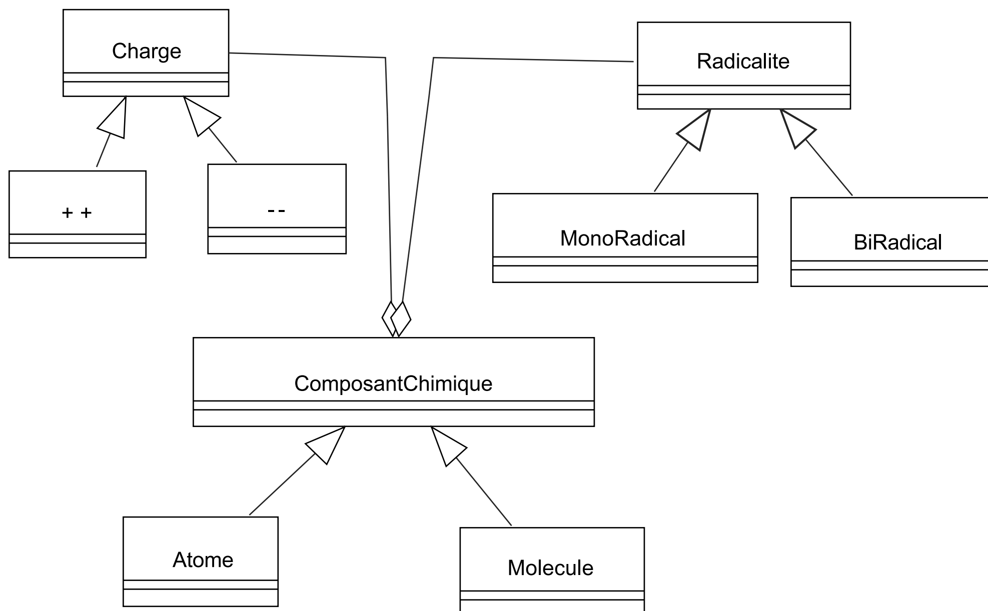
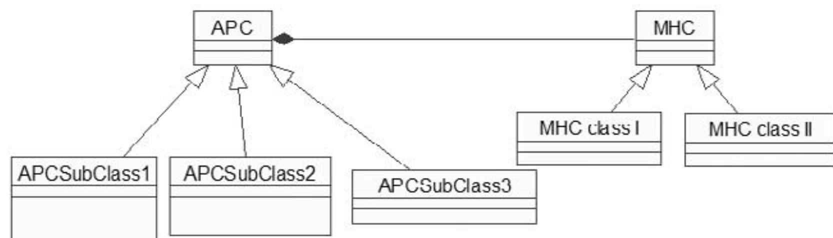


Figure 23-14
Le pattern « pont ».

Figure 23-15
Le pattern pont appliqué à l'immunologie



Index

Numériques

2-tier 388
3-tier 389
.Net 142, 376, 407, 489

A

abstract 303
abstract factory 591
abstraction 300
Access 483
acteur 62
ActionListener en Java 365
adressage indirect 14, 96
affectation d'objet 347
agrégation 180
analyse
  objet 62
  procédurale 60
annuaire 406, 419
argument 68, 179
  objet 95, 140
  passage par référent 90, 93
  passage par valeur 89, 93
assemblage 83, 130
assignation d'objet 349
association de classes 67, 175
atome 570
attribut 10, 67
  private 111, 112
  public 111
automate cellulaire 426

B

base 266
base de données 389, 437, 483
  objet 66
  OO 504, 505
  relationnelle 13, 483
  relationnelle-objet 66, 495, 500
Berners-Lee, Tim 408
biologie 66, 426
Booch, Gary 160

C

cardinalité 175
cas d'utilisation 160
casting 94, 281, 328, 479
  critique 284
  explicite 218
  implicite 218, 281
cerveau 425
charte du bon programmeur OO 110, 131
chimie 426
  computationnelle 566, 583, 585, 590
cinétique réactionnelle 579
classe
  abstraite 255, 301, 303, 308, 363
  association 67, 79, 169, 175
  dépendance 68, 95, 179
  enceinte de confinement 120
  fichier 69, 76, 179
  généralité 19
  imbrication 128
  interaction 74
  Object 326, 329, 479, 549
  observable 451
  sans objet 300
  Thread 429
  Vector 327, 328, 539, 540
clé
  étrangère 491
  primaire 485
clonage d'objet 101, 336
CLR (Common Language runtine) 143
CLS (Common Language Specification) 143
collection 220
comparaison d'objets 349
compilation 67, 74, 76, 111, 218, 229, 243, 280, 281, 300, 365, 373, 549
composition 195
  d'objets 17, 180, 219
compteur de référents 148, 151
conception objet 62
constructeur
  héritage 234
  par copie 101, 140, 348, 353
cookie 418
Corba 160, 391, 397, 407
  invocation
    dynamique 405
    statique 405
  services 401

D

Dahl, Ole-Johan 58
Deittel et Deittel 75
délégué 454
  en C# 432
delete 145, 147, 187, 204
dépendance
  de classe 68, 95, 179
  fonctionnelle versus indépendance logicielle 63, 64
design pattern 396, 450
destructeur 140, 185, 188, 190

diagramme
  d'objet 178
  de classe 161, 165, 198, 212, 255, 313, 370, 540, 567
  de composant 179, 379
  de séquence 161, 165, 199, 434
  UML 161
disque dur 69, 136, 470
dll 77
DTD (Data Type Dictionnary) 408, 409

E

écosystème 60, 196, 212, 301, 424
effet papillon 132
égalité de deux objets 331
encapsulation 102, 112, 115, 131, 230, 231, 363
  des méthodes 124
espace de nommage 130, 399

F

factory_method 591
fichiers
  .cpp en C++ 378, 379
  .h en C++ 379
finalize 148, 181
flux filtré 471
fonctionnement de type yoyo 229
friend 127, 350
fuite de mémoire 146

G

Gamma, Helm, Johnson et Vlissides 591
Gang des quatre 396, 591
garbage collector 150, 154
Gates, Bill 142, 311
Gemstone 504
génération automatique de code 198
généricité 94, 546
gestion
  d'exception 118, 282, 338, 392, 473
  mémoire 139, 144, 173
Gnutella 390
Gosling, James 74
graphe 536, 555
  moléculaire 574

H

Hejlsberg, Anders 142
héritage
  addition de propriétés 215
  arbre versus graphe 238
  d'interfaces 363
  des attributs 213
  des méthodes 220
  des streams 471
  du constructeur 234
  économie 220
  généralités 24, 26, 64, 198, 493
  interprétation ensembliste 218, 264
  justification 220, 255
  mécanisme 213
  public en C++ 237
  versus composition 219, 516
  virtuel en C++ 247
Hopfield, John 537
HP 410
HTML 409
HTTP 408

I

IBM 399, 410
IDL (Interface Definition Langage) 398
IDLJ (Interface Definition Langage Java) 399
imbrication de classe 128, 185
include 380
indépendance logicielle 63, 64, 118
informatique
  séquentielle 425
  ubiquitaire 391
Informix 483
instance 19
intégrité référentielle 491
intelligence artificielle 217
interface 102, 124, 127, 245, 338, 363, 392, 399, 480
  fichier 371
  graphique 309
  multihéritage 370
  Observer 452
  versus implémentation 124, 363
  versus implémentation en C++ 379
Internet 104, 382, 388, 435
Iona 399

J

Jacobson, Ivar 160
Java 74
JAX-RPC 407
JDBC 487
Jini 75, 406, 419
Jobs, Steve 310
JXTA 390

K

Kant 217
Kauffman, Stuart A. 131, 567
Kay, Alan 310, 311, 566
Kazaa 390
KeyListener en Java 364, 368
Khü n, Thomas 217

L

leasing 406
liaison 573
LIFO (Last-In First-Out) 139
LISP 152
liste liée 539
lookup 406

M

machine de von Neumann 426
match de football 196, 256
mé thode
  private 124
  run 429
mémoire
  associative 537
  cache 137
  compactage 152
  fuite 146
  pile 89, 138, 169, 173, 185, 187, 227, 349, 539
  RAM 18, 136, 137, 229, 343, 470
  tas 144, 169, 173, 187, 227, 275, 308, 347, 539
message 102, 174
  asynchrone 434
  envoi 70, 169, 390, 404
  généralités 22, 66
  synchrone 434

méthode
abstraite 302, 363
appel 20, 82
argument 69
d'accès 113
définition 19, 22
main 66
message 102
private 124
public 124
redéfinition 254
signature 102
versus message 102
virtuelle 275
pure 303
Meyer, Bertrand 70, 143
Microsoft 160, 407
Minsky, Marvin 217, 326, 537
molécule 570
MouseListener en Java 364
multihéritage 238, 370, 523
d'interfaces 245
en C++ 239
multitâche 428
multithreading 200, 330, 364
définition 427
généralités 139
join 444
par composition 432
par héritage 432
priorité 436
sleep 436
start 432
suspension 435
synchronisation 435, 437
multi-tier 389

N

Napster 390
new 137, 144, 300, 303
nœud 536
null 183
Nygaard, Kristen 58

O

ObjectStore 504
objet
affectation 347
clonage 336, 339, 341
compilation 195
composite 16
composition 67
cycle de vie 18, 144
dépendance 450
distribué 104, 388, 391, 435
état 17
intégrité 115
interaction 21, 64, 66, 127
persistance 470, 471
référent 13
sauvegarde 353
test d'égalité 331
versus procédural 60, 82, 118, 144
ODBC (Open DataBase Connectivity) 487
ODMG (Object Data Management Group) 504
OleDb 489
OMG (Objet Management Soap) 160, 398
OMT (Object Modelling Technique) 160
OOD (Object Oriented Design) 160
OOSE (Object Oriented Software Engineering) 160
OQL (Objet Query Language) 501
OQL(Objet Query Language) 504
Oracle 483, 501
OrbixWeb 399
override 262, 266, 306, 375

P

package 83, 130, 399
Papert, Seymour 311
parallélisme 426
passage
par référent 90, 93, 99
par valeur 89, 99
pattern
bridge 591
façade 591
memento 592
observer 591
proxy 591
visitor 592
peer-to-peer 390
Piaget, Jean 217
Poet 504
pointeur 90, 140, 169
polymorphisme 245, 255, 375, 522
Corba 404
et casting 285
généralités 26, 102, 309
mécanisme 256, 280
par défaut 272
RMI 404
principe
de localité 137
de substitution 216, 218, 271, 281
private 111, 118, 127
programmation événementielle 450
protected 130, 230, 231
proxy 406, 412, 591
public 111, 118, 127

R

ramasse-miettes 96, 150, 154, 181, 190, 350
Rational 160
Rose 160, 174, 239, 351
réacteur chimique 567
réactions chimiques 577
redéfinition des méthodes 254, 263, 364
référent 13, 95, 144, 148, 169, 174, 276, 333, 373, 432
en C++ 90
fou 146, 148
registre 393, 396
relation
1-n 490
n-n 492
répertoire 85
réseau 536
de Hopfield 537
de neurones 217, 426, 536
génétique 536
immunitaire 536
réutilisation 220
Riel, Arthur J. 110
RMI (Remote Method Invocation) 75, 389, 391, 407
robustesse 391
rôles 175
RTTI (Run-Time Type Information) 260
Rumbaugh, James 160
Runnable en Java 364

S

sauvegarde d'objet 353
schéma XML 408, 409
sciences cognitives 27, 537
sémaphore 442
sérialisation 478
services Web 143, 407, 410
seti@home 388
signature 245, 363
  de méthode 124
Simula 58, 310
singleton 591
skeleton 395, 396
Smalltalk
  voir aussi Squeak 310
Soap 411, 413
sous-classe 213
SQL 483
SQL3 500, 502
Squeak
  voir aussi Smalltalk 311
squelette de code 198
stabilité du logiciel 118, 120, 125, 363, 382
streams 471
Stroustrup, Bjarne 75, 93, 284, 546
structure en C# 142, 169, 185, 227, 278, 353, 376
stub 395, 396, 399
Sun 388, 407, 410
super 234, 264
superclasse 212
surcharge d'opérateur 347, 349, 476
Sybase 483
synchronisation des threads 437
système
  complexe 66, 131, 312, 537
  d'exploitation 309, 427

T

table 484
  virtuelle en C++ 275
tableau 538
  d'objets 302, 327
  en C++ 269
taxonomie 214, 255
Taylor, David A. 66
TCP/IP 382
template 546, 549
temporisation 471
thread 427
TogetherJ 165, 198, 200, 239
traitement en surface et en profondeur 343
trio 10
trois amigos 160
typage
  dynamique 268, 274, 281, 300
  statique 268, 274, 281
  statique versus typage dynamique 271, 305
type
  entier 20
  primitif 11, 67

U

UDDI (Universal Description Discovery and Integration) 419
UML 160, 161, 164, 220, 312, 351, 434, 549, 566
  avantages 196

V

valeur 93
Versant 504
vie artificielle 426
virtual 260, 261, 273, 375
virtual/override 278
Visibroker 399
Visual Studio .Net 142
von Neumann, John 426

W

W3 408
Walter Fontana 566, 567
WDSL 410, 419
Web sémantique 408
WebSphere 399
Windows 311
Wittgenstein 25

X

Xerox PARC 311
XML 407, 409

Hugues Bersini
Ingénieur physicien, directeur du Laboratoire d'intelligence artificielle de l'Université libre de Bruxelles, Hugues Bersini enseigne l'informatique et la programmation aux facultés polytechnique et Solvay de cette même université.
Ivan Wellesz est développeur Java indépendant et formateur Unix, C et Java chez Orsys. Il a travaillé treize ans chez Tektronix où il a participé à la conception d'interfaces homme-machine et de systèmes d'armes à DCN-BREST.

L'approche objet est enseignée dans les universités dès les premiers niveaux des cursus informatiques, car sa compréhension est le prérequis indispensable à toute pratique économe, fiable et élégante de la plupart des techniques informatiques qui en sont dérivées, depuis Java et Python, jusqu'à UML 2, en passant par C# et C++.

L'objet par la pratique avec Python, Java, C# et C++ et PHP 5… en UML 2

Cette quatrième édition de l'ouvrage *L'orienté objet* décortique l'ensemble des mécanismes de la programmation objet (classes et objets, interactions entre classes, envois de messages, encapsulation, héritage, polymorphisme, modélisation…) en les illustrant d'exemples empruntant aux technologies les plus populaires : Java et C#, C++, Python, PHP 5, UML 2, mais aussi les services web, RMI, les bases de données objet, différentes manières de résoudre la mise en correspondance relationnel/objet par le langage innovant de requête objet LINQ et enfin les design patterns. Chaque chapitre est introduit par un dialogue vivant, à la manière du maître et de l'élève, et se complète de nombreux exercices en UML 2, Java, Python, PHP 5, C# et C++.

À qui s'adresse ce livre ?

- Ce livre sera lu avec profit par tous les étudiants de disciplines informatiques liées à l'approche objet (programmation orientée objet, modélisation UML, Java, Python, PHP 5, C#/C++…) et pourra être utilisé par leurs enseignants comme matériel de cours.
- Il est également destiné à tous les développeurs qui souhaitent approfondir leur compréhension des concepts objet sous-jacents au langage qu'ils utilisent.

Au sommaire

Principes de base : L'objet, version passive, version active – Notion de classe – Interaction et hiérarchie des objets – Polymorphisme – Héritage. **La classe, module fonctionnel et opérationnel** – La classe garante de son bon usage – Premier programme complet en Java, PHP 5 et Python, C# et C++. **Du procédural à l'orienté objet :** Mise en pratique – Analyse – Conception. **Les objets parlent aux objets :** Envois de messages – Association de classes – Dépendance de classes. **Collaboration entre classes :** Compilation Java et effet domino – En PHP 5, Python, C# et en C++ – Association unidirectionnelle/bidirectionnelle – Auto-association – Assemblage. **Méthodes ou messages :** Passage d'arguments prédéfinis dans les messages – Arguments objets. **L'encapsulation des attributs :** Accès aux attributs d'un objet. **Les classes et leur jardin secret :** Encapsulation des méthodes – Niveaux intermédiaires d'encapsulation – Éviter l'effet papillon. **Vie et mort des objets :** C++, ou le programmeur seul maître à bord – Java, PHP 5, Python et C#, ou la chasse au gaspi. **UML :** Représentation graphique standardisée – Diagrammes de classe et de séquence. Héritage – Regroupement en superclasses – Héritage des attributs – Composition : Héritage des méthodes – Encapsulation protected – Héritage public en C++ – Multihéritage. **Redéfinition des méthodes :** Un match de football polymorphique. **Abstraite, cette classe est sans objet :** De Canaletto à Turner – Abstraction syntaxique – Supplément de polymorphisme. **Clonage, comparaison et assignation d'objets :** La classe Object – Égalité, clonage et affectation d'objets en C++ et C#. **Interfaces :** Favoriser la décomposition et la stabilité – Java, PHP 5 et C# : interface via l'héritage – C++ : fichiers .h et .cpp. **Distribution d'objets sur le réseau :** RMI – Corba – Services web. **Multithreading :** Implémentation en Java, PHP 5, Python et C# – Multithreading et diagrammes de séquence UML – Vers les applications distribuées – Des threads équirépartis et synchronisés. **Programmation événementielle :** Des objets qui s'observent – En Java, PHP 5, Python et C#. **Persistance d'objets :** Sauvegarde sur fichier – La sérialisation – Bases de données relationnelles et objet – La bibliothèque LINQ. Simulation d'un flipper. **Les graphes :** Liste liée – Généricité en C++ et en Java, PHP 5 et Python – Les design patterns.

Le code source des exercices et leurs corrections sont fournis sur le site d'accompagnement www.editions-eyrolles.com.