Jouer au morpion [dx02] - Exercice résolu

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 20 mai 2018

Table des matières

C - Jouer au morpion (Solution)

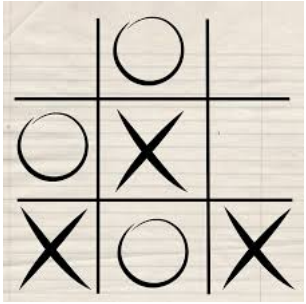
Mots-Clés Exercices généraux ■
Requis Axiomatique impérative ■
Difficulté ●●○ (3 h) ■

Objectif
Cet exercice fait jouer deux joueurs humains au jeu du morpion.
Il initie à :

- L'écriture de la **documentation** et sa génération HTML.
- La notion de **contrat des opérations**.

Il revient sur :

- L'analyse descendante (méthode de décomposition).
- La stratégie par étapes (tests des opérations).



Jeu du morpion (google/images)

...(énoncé page suivante)...

1 Énoncé

Jeu du morpion

Appelé aussi **tic-tac-toe**, le **morpion** est un jeu de pions se jouant à deux sur un damier de neuf cases (3×3). Chaque joueur pose à tour de rôle l'un de ses pions. Le vainqueur est celui qui réussit à aligner trois de ses pions en ligne, colonne ou diagonale.

Exemple

La figure ci-dessous rapporte quelques étapes d'une partie où l'un des joueurs a les pions `X` (joueur qui commence) et l'autre joueur les pions `O`. Le dernier tour montre la victoire du joueur ayant les pions `X`.

```
. . .      O . .      O . .      O O .      O O X
. . .      . . X      O X X      O X X      O X X
. . .      . . .      . . .      X . .      X . .
```

Objectif

Développer un programme qui fait jouer deux joueurs humains.
Le programme doit à tour de rôle :

- Demander les coordonnées du pion au joueur actif.
- Remplir la case correspondante après avoir vérifié qu'il s'agit d'un emplacement du plateau non encore occupé.
- Tester la fin de jeu.
- Sinon, passer la main à l'autre joueur.

Le jeu se termine si un des joueurs gagne ou s'il n'y a plus de cases libres pour jouer (partie nulle).

Analyse

Les étapes sont dans l'ordre :

1. Le cahier des charges (objectifs théoriques et produits attendus).
2. Les structures de données et les opérations.
3. L'algorithme du programme principal.
4. L'analyse descendante de chaque opération.

2 Analyse

2.1 Cahier des charges

Fonctionnement du programme
Il se déduit du problème.
Ici c'est :

1. Création et initialisation d'un plateau de jeu
2. Affichage du tableau initial
3. Assignation d'un symbole pour chaque joueur
4. Décision du joueur qui commence
5. Boucle d'interaction avec les joueurs :
   (a) On fait jouer le joueur actif
   (b) On affiche le plateau modifié avec le coup du joueur
   (c) On teste la fin du jeu et sinon c'est au tour de l'autre joueur
6. Affichage du gagnant éventuel

Hypothèses et contraintes
Ce sont :

- Le tableau est de taille 3×3.
- S'il n'y a plus de possibilité de gagner pour aucun joueur, le tableau doit être totalement rempli afin de terminer.

Remarque
Dans une version améliorée, on pourra tester si un tableau est tel qu'aucun joueur ne peut plus gagner.

Cas d'erreurs
On en dénombre trois :

1. Les coordonnées d'un pion sont en dehors du tableau.
2. Il n'y a plus d'emplacement où jouer.
3. On veut jouer sur une case déjà cochée.

Nous ne traitons pas les erreurs lors de la saisie des coordonnées.

...(suite page suivante)...

2.2 Structures de données

Structures de données

Ce sont :

- Un plateau de jeu : matrice 3×3 où sont enregistrés les coups.
- Le joueur qui joue : pour pouvoir enregistrer son coup dans le plateau.

Première modélisation

```
Constante NDIM <- 3
Typedef Grille = Entier[NDIM,NDIM] # le plateau
Variable quiJoue : Entier # le prochain a jouer
```

- Un joueur : modélisé par un entier `1` ou `2`
- Le plateau : une grille 3×3 d'entiers où
  - Case avec 0 <=> case vide
  - Case avec 1 <=> case cochée par le joueur 1
  - Case avec 2 <=> case cochée par le joueur 2
- Propriétés à garantir à tout moment :
  - Joueur dans $\{1, 2\}$
  - Le plateau est rempli uniquement avec $\{0, 1, 2\}$
- On attribue un symbole par joueur (`1: 'x', 2:'o'`) pour les besoins d'affichage.
- Avantage : Nul besoin d'une opération d'initialisation pour le plateau (selon les langages) car la matrice est initialisée à 0 par défaut et 0 <=> case vide.
- Inconvénient : Lors de l'affichage du plateau, il faut transformer chaque entier dans $\{0, 1, 2\}$ vers le symbole correspondant `{' ','x','o'}`.

Modélisation alternative

```
Constante NDIM <- 3
Typedef Grille = Caractère[NDIM,NDIM] # le plateau
Variable quiJoue : Caractère # le prochain a jouer
```

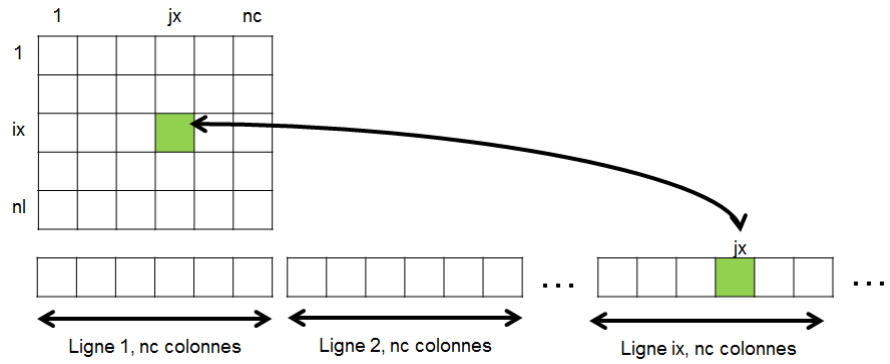
- Un joueur : modélisé par un caractère `'x'` ou `'o'`
- Le plateau : une grille 3×3 de caractères rempli par `{' ','x','o'}`
- Avantage : Nul besoin d'opération de conversion lors de l'affichage.
- Inconvénient : Nécessité d'une opération d'initialisation du plateau avec `' '` dans toutes les cases.

Autre modélisation

Un stockage linéaire du tableau bidimensionnel selon l'une des deux modélisations précédentes (entier ou caractère). Par exemple, en prenant la première :

```
Constante NDIM <- 3
Typedef Grille = Entier[NDIM * NDIM] # le plateau
Variable quiJoue : Entier # le prochain a jouer
Fonction indexTab2d(ix : Entier; jx : Entier) : Entier
```

La fonction permet de transformer une position ligne `ix` et colonne `jx` en un indice dans une `Grille` :



Conclusion

Les modélisations sont équivalentes :

- En efficacité, et
- En facilité d'utilisation.

Nous utiliserons la première.

Traduisez la modélisation dans votre langage de programmation.

Solution simple

Il est intéressant d'écrire deux opérations (accès et modifieur) de sorte à devenir plus ou moins indépendant de la modélisation.

Validez votre code avec la solution.

Solution C @[pgmorpion.c]

...(suite page suivante)...

2.3 Opérations du programme

Nous pouvons déduire les opérations à partir de la description du @[Fonctionnement du programme] dans la section @[Cahier des charges].

Opérations du programme

Ici on a :

1. Création et initialisation d'un plateau de jeu `gr`
   ==> `initialiserPlateau(gr)`
2. Affichage des messages de bienvenue et
   Assignation d'un symbole pour chaque joueur
   ==> `afficherBienvenue`
3. Affichage du plateau `gr`
   ==> `afficherPlateau(gr)`
4. Décision du joueur qui commence
   ==> `quiJoue <- 1`
5. Boucle d'interaction avec les joueurs :
   (a) On fait jouer le joueur actif sur le plateau `gr`
   ==> `jouer(quiJoue,gr)`
   (b) On affiche le plateau modifié avec le coup du joueur
   ==> `afficherPlateau(gr)`
   (c) On teste la fin du jeu sur le plateau `gr` :
   - Le joueur actif a-t-il gagné ?
     ==> `victoireDe(quiJoue,gr)`
   - Le jeu est-il bloqué ?
     ==> `plateauBloque(gr)`

   (d) Sinon c'est le tour de l'autre joueur
   ==> `quiJoue <- joueurSuivant(quiJoue)`
   (e) Et on recommence
6. Affichage du gagnant éventuel
   ==> Opération différée en 5.(c)

...(suite page suivante)...

2.4 Ébauche du programme

Afin de valider la liste d'opérations, la prochaine étape consiste à écrire une procédure de jeu en utilisant les opérations définies ci-avant :

```
Action jeuMorpion
Variable gr : Grille
Variable quiJoue : Entier
Variable finJeu : Booléen
Début
  |  quiJoue <- 1
  |  initialiserPlateau ( gr )
  |  afficherBienvenue
  |  afficherPlateau ( gr )
  |  finJeu <- Faux
  |  TantQue ( Non finJeu ) Faire
  |    |  jouer ( quiJoue , gr )
  |    |  afficherPlateau ( gr )
  |    |  Si ( victoireDe ( quiJoue , gr ) )
  |    |    |  Afficher ( quiJoue , " gagne" )
  |    |    |  finJeu <- Vrai
  |    |  Sinon Si ( plateauBloque ( gr ) ) Alors
  |    |    |  Afficher ( "Jeu bloque" )
  |    |    |  finJeu = Vrai
  |    |  Sinon
  |    |    |  quiJoue <- joueurSuivant ( quiJoue )
  |    |  FinSi
  |  FinTantQue
Fin
```

Traduisez cette ébauche dans votre langage de programmation.

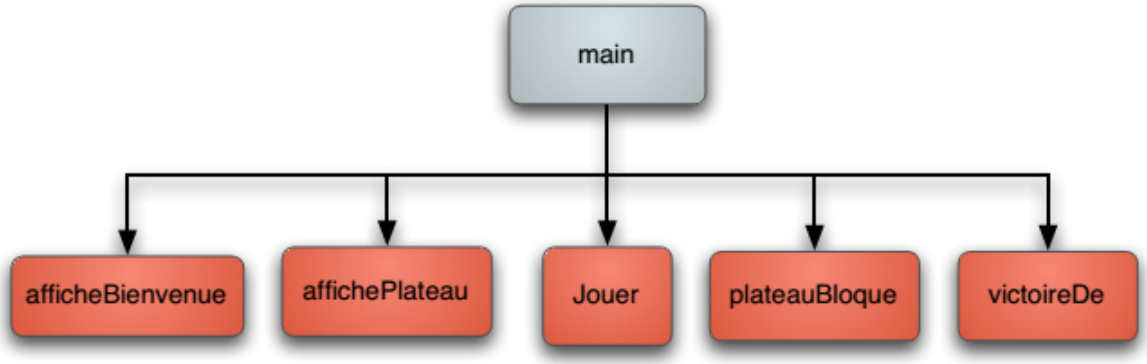
Remarque

Le programme obtenu est une **ébauche** : il utilise des opérations qui n'existent pas encore. Chacune de ces opérations est un sous-problème que nous pouvons traiter de manière similaire au problème global.

...(suite page suivante)...

Opérations du programme

Le diagramme ci-dessous montre les opérations utilisées par le programme qu'il faudra raffiner en procédures/fonctions. Elles-mêmes pourront utiliser d'autres opérations (nouvelles branches dans le diagramme).



...(suite page suivante)...

3 Analyse descendante

Analyse descendante
Pour chacune des opérations, nous allons déterminer :

- Quelles sont ses entrées ?
- Quelles sont ses sorties ?
- Procédure ou fonction ?
- Sa signature (dit aussi profil) ?
- Les contraintes, hypothèses, cas d'erreur ?
- Son fonctionnement général ?
- Les données et opérations nécessaires ?
- Son ébauche en termes de ces opérations ?

Il est possible que d'autres opérations soient nécessaires.
Cela nous fera « descendre » d'un niveau dans le raffinement.

Écrivez le profil...
Ceci signifie :

« Écrivez le profil **ainsi que** sa partie documentation »

Profil documenté (fonction ou procédure)

```
/**
  ...
  @param[in,out] nomparam - description
  @return description
*/
... nomModule(listeDesParametres) ...
```

3.1 Opération initialiserPlateau

L'opération `initialiserPlateau` initialise le plateau à vide.
Déterminez :

1. Entrants, Sortants et Mixtes ?
2. Procédure ou fonction ?
3. Signature ?
4. Contraintes, hypothèses, cas d'erreur ?
5. Données et opérations nécessaires ?

Solution Profil

1. Sortants : Un plateau dont les cellules doivent être initialisées à vide
2. Procédure
3. `Action initialiserPlateau(R gr : Grille)`
4. Aucune
5. Aucune

Écrivez l'opération `initialiserPlateau(gr)`.

Validez votre procédure avec la solution.

Solution C @[pgmorpion.c]

```
void initialiserPlateau(Grille gr)
{
  int j, k;
  for (j = 0; j < NDIM; ++j)
  {
    for (k = 0; k < NDIM; ++k)
    {
      gr[j][k] = 0;
    }
  }
}
```

3.2 Opérations d'affichage

L'opération `afficherBienvenue` affiche des messages de bienvenue et les symboles des joueurs. Déterminez :

1. Entrants, Sortants, Mixtes ?
2. Procédure ou fonction ?
3. Signature ?
4. Contraintes, hypothèses, cas d'erreur ?
5. Données et opérations nécessaires ?

Solution Profil

1. Aucun
2. Procédure
3. `Action afficherBienvenue`
4. Aucune
5. `symboleDe` qui convertit un entier $(0, 1, 2)$ en un symbole à afficher (`'.'`, `'x'` ou `'o'`)

Écrivez l'opération `afficherBienvenue` de sorte qu'elle affiche :

```
Bienvenue au jeu du morpion
---------------------------
Joueur 1 = x et Joueur 2 = o
Plateau initial
```

(Les symboles `x` et `o` sont affichés par la fonction `symboleDe` définie ci-après.)

Pour l'opération `symboleDe`, qui affiche le symbole du joueur, déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Contraintes, hypothèses, cas d'erreur ?

Solution Profil

1. Un entier dans [0..2]
2. Un caractère `'.'`, `'x'` ou `'o'`
3. Fonction
4. `Fonction symboleDe(j : Entier)`
5. `j` doit être compris dans [0..2]

Écrivez l'opération `symboleDe(j)`.

L'opération `afficherPlateau` affiche le plateau.
Déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Contraintes, hypothèses, cas d'erreur ?

Solution Profil

1. Un plateau
2. Aucune
3. Procédure
4. `Action afficherPlateau(gr : Grille)`
5. La grille a été initialisée et/ou modifiée

Écrivez l'opération `afficherPlateau(gr)` de sorte qu'elle affiche le plateau comme suit :

```
xy 0  1  2
----------
0| .  .  .
1| .  .  .
2| .  .  .
```

Validez votre fonction et vos procédures avec la solution.

Solution C @[pgmorpion.c]

```c
char symboleDe(int j)
{
  const char symboles[] = {'.', 'x', 'o'};
  return symboles[j];
}
```

```c
void afficherBienvenue()
{
  printf("Bienvenue au jeu du morpion\n");
  printf("---------------------------\n");
  printf("Joueur 1 = %c -- ",symboleDe(1));
  printf("Joueur 2 = %c\n",symboleDe(2));
  printf("\n");
  printf("Plateau initial: \n");
}
```

```c
void afficherPlateau(const Grille gr)
{
  printf("\n");
  printf("xy 0  1  2\n");
  printf("----------\n");
  int j, k;
  for (j = 0; j < NDIM; ++j)
  {
    printf("%d|",j);
    for (k = 0; k < NDIM; ++k)
    {
       printf(" %c ",symboleDe(gr[j][k]));
    }
    printf("\n");
  }
  printf("\n");
}
```

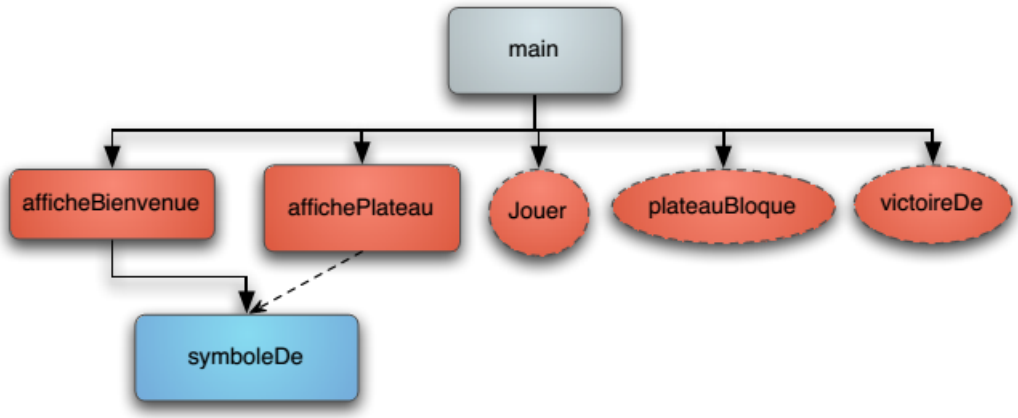
Solution commentée

Afin de rendre plus simple la conversion, la fonction `symboleDe` utilise une donnée interne : un tableau de caractères pour effectuer la correspondance. Hypothèse : `j` dans $\{0, 1, 2\}$.

3.3 Mise en oeuvre par étapes

L'arbre du raffinement

Le diagramme montre ce que nous avons développé jusque-là (dans les carrés) et ce qui reste à faire (dans les cercles) :



Stratégie par étapes

Il est utile d'établir une stratégie de mise en oeuvre du programme en machine.

- Une fois une procédure/fonction écrite, nous procèderons à son test.
- Une fois les tests réussis, nous l'ajoutons au programme.

Pour les tests, il est intéressant de commencer par écrire les procédures d'initialisation et d'affichage ce que nous avons fait. D'où :

Dans votre procédure `jeuMorpion`, ne testez que les affichages initiaux :

```
Début
  |  quiJoue <- 1
  |  initialiserPlateau( gr )
  |  afficherBienvenue
  |  afficherPlateau( gr )
Fin
```

Testez. Résultat d'exécution :

```
Bienvenue au jeu du morpion
---------------------------
Joueur 1 = x -- Joueur 2 = o

Plateau initial:

xy 0  1  2
----------
```

```
0| .  .  .
1| .  .  .
2| .  .  .
```

Générez la documentation HTML de votre programme.

3.4 Opération jouer

L'opération `jouer` demande les coordonnées à jouer et coche la case avec le numéro du joueur. Si les coordonnées sont invalides, on demande une nouvelle saisie.
Déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Contraintes, hypothèses, cas d'erreur ?
6. Données et opérations nécessaires ?

Solution Profil

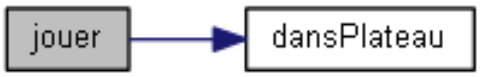
1. Un entier (joueur), un plateau
2. Aucune, mais il y a **modification** du plateau
3. Procédure
4. `Action jouer(j : Entier; DR gr : Grille)`
5. On a :
   - `j` doit être dans [1..2]
   - Les cases de `gr` doivent contenir des entiers dans [0..2]
   - `gr` a au moins une case vide (non plein)
   - Erreur : coordonnées invalides si en dehors du plateau ou si elles correspondent à une case déjà cochée.
6. Utilise `dansPlateau` qui teste si les coordonnées à jouer sont dans les bornes du plateau

Écrivez une fonction `dansPlateau(x,y,gr)` qui teste si la coordonnée en `(x,y)` (ligne,colonne) est ou non dans le plateau `gr`.

Orientation

A cause de notre modélisation, le paramètre `gr` n'est pas utile pour cette fonction : la grille est représentée par un tableau bidimensionnel et non comme une structure. Il est cependant essentiel de le conserver. En effet : une modification de structures de données **ne doit jamais entraîner une modification de signature** d'une (ou plusieurs) opération.

Écrivez alors l'opération `jouer(j,gr)`.



Validez votre fonction et procédure avec la solution.

Solution C @[pgmorpion.c]

```
bool dansPlateau(int x, int y, const Grille gr)
{
  return (0 <= x && x < NDIM && 0 <= y && y < NDIM);
}
```

```
void jouer(int j, Grille gr)
{
  bool ok = false;
  printf("Le tour au joueur %d\n",j);
  while (!ok)
  {
    printf("Coordonnees x y? ");
    int x, y;
    scanf("%d%d",&x,&y);
    if (!dansPlateau(x,y,gr))
    {
      printf("Coordonnees invalides hors plateau\n");
    }
    else if (gr[x][y] != 0)
    {
      printf("Case occupee\n");
    }
    else
    {
      gr[x][y] = j; // on coche la case
      ok = true;
    }
  }
}
```

3.5 Opération joueurSuivant

L'opération `joueurSuivant` donne la main au joueur suivant.
Déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Contraintes, hypothèses, cas d'erreur ?
6. Données et opérations nécessaires ?

Solution Profil

1. Un entier (joueur)
2. Aucune, mais il y a **modification** du numéro de joueur
3. (Au choix. Nous utiliserons une) Fonction
4. `Fonction joueurSuivant(j : Entier) : Entier`
5. `j` doit être dans [1..2]
6. Aucune

Écrivez l'opération `joueurSuivant(j)`.

Validez votre fonction avec la solution.

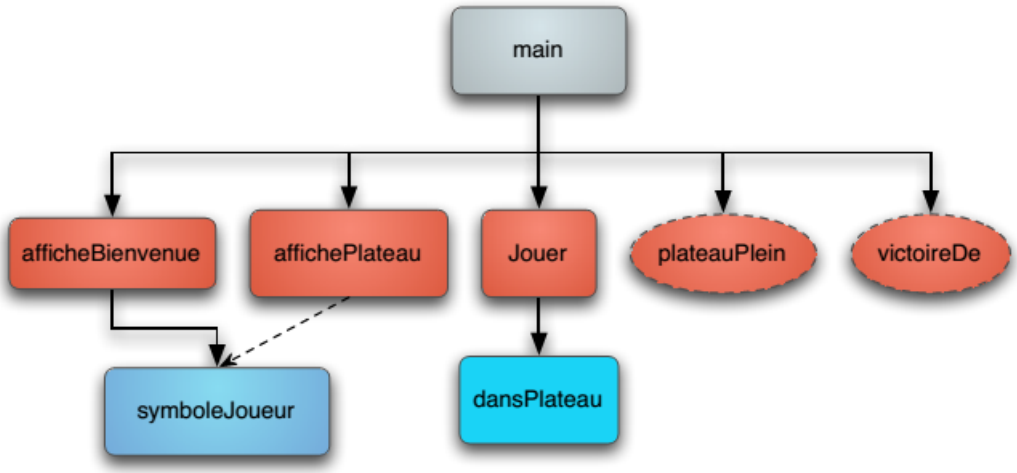
Solution C @[pgmorpion.c]

```
int joueurSuivant(int j)
{
  return (j == 1 ? 2 : 1);
}
```

3.6 Les tours du jeu

L'arbre du raffinement

Le diagramme montre ce que nous avons développé jusque là (dans les carrés) et ce qui reste à faire (dans les cercles) :



Les tours du jeu

A ce point de l'exercice, nous pouvons tester le comportement de ces opérations.

```
Début
  |  ...
  |  TantQue (Non finJeu) Faire
  |    |  jouer( quiJoue , gr )
  |    |  afficherPlateau( gr )
  |    |  quiJoue <- joueurSuivant( quiJoue )
  |  FinTantQue
Fin
```

Complétez votre procédure.

Testez. Exemple d'exécution :

```
Le tour au joueur 1
Coordonnees (x,y)? ( 1 , 2 )

xy 0  1  2
----------
0| .  .  .
1| .  .  x
2| .  .  .

Le tour au joueur 2
Coordonnees (x,y)? ( 1 , 0 )

xy 0  1  2
----------
0| .  .  .
1| o  .  x
2| .  .  .
```

3.7 Opération plateauBloque

L'opération `plateauBloque` teste si le jeu est bloqué avec toutes les cases cochées. Déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Contraintes, hypothèses, cas d'erreur ?

Solution Profil

1. Un plateau
2. Un booléen
3. Fonction
4. `Fonction plateauBloque(gr : Grille) : Booléen`
5. Hypothèses : `gr` n'a pas de configuration gagnante. L'appel à cette opération n'a de sens que si le joueur ne vient pas de gagner.

Écrivez l'opération `plateauBloque(gr)`.

Validez votre fonction avec la solution.

Solution C @[pgmorpion.c]

```c
bool plateauBloque(const Grille gr)
{
  int j, k;
  for (j = 0; j < NDIM; ++j)
  {
    for (k = 0; k < NDIM; ++k)
    {
      if (gr[j][k] == 0)
      {
        return false;
      }
    }
  }
  return true;
}
```

Complétez votre procédure.

Testez. Exemple d'exécution (Fin par blocage) :

```
Le tour au joueur 2
Coordonnees (x,y)? ( 2 , 2 )

xy 0  1  2
----------
0| x  o  x
1| x  o  .
2| o  x  o

Le tour au joueur 1
Coordonnees (x,y)? ( 1 , 2 )

xy 0  1  2
```

```
----------
0| x  o  x
1| x  o  x
2| o  x  o

Jeu bloque. Fin sans gagnant
```

3.8 Opération victoireDe

Enfin l'opération `victoireDe` teste si une ligne, colonne ou diagonale est remplie par le numéro du joueur.
Déterminez :

1. Entrants ?
2. Sortants ?
3. Procédure ou fonction ?
4. Signature ?
5. Données et opérations nécessaires ?

Solution Profil

1. Un numéro de joueur, un plateau
2. Un booléen
3. Fonction
4. `Fonction victoireDe(j : Entier; gr : Grille) : Booléen`
5. On peut :
   - Soit écrire **directement** les 8 possibilités
   - Soit définir une fonction `alignement(j,ix,jx,iincr,jincr,gr)` qui teste l'alignement du joueur `j` sur le plateau `gr`.

Écrivez l'opération `victoireDe(j,gr)`.

Orientation

Éventuellement, écrivez une fonction `alignement(j,ix,jx,iincr,jincr,gr)` qui teste et renvoie `Vrai` si il y a un alignement du joueur numéro `j` sur un plateau `gr`, `Faux` sinon. La position de la première case est `(ix,jx)` et l'incrémentation en ligne et colonne est `(iincr,jincr)`.

```
.      jincr ->
---------------------------
|(ix,jx)|        |        |  iincr
---------------------------    |
|       |        |        |    V
---------------------------
|       |        |        |
---------------------------
```

Validez votre fonction avec la solution.

Solution C @[pgmorpion.c]

```
bool alignement(int j, int ix, int jx, int iincr, int jincr, const Grille gr)
{
  bool b = true;
  int k;
  for (k = 0; k < NDIM && b; ++k)
  {
     b = b && gr[ix][jx] == j;
     ix += iincr;
     jx += jincr;
  }
  return b;
}
```

```
bool victoireDe(int j, const Grille gr)
{
  return (gr[0][0]==j && gr[0][1]==j && gr[0][2]==j) // ligne 0
  || (gr[1][0]==j && gr[1][1]==j && gr[1][2]==j) // ligne 1
  || (gr[2][0]==j && gr[2][1]==j && gr[2][2]==j) // ligne 2
  || (gr[0][0]==j && gr[1][0]==j && gr[2][0]==j) // colonne 0
  || (gr[0][1]==j && gr[1][1]==j && gr[2][1]==j) // colonne 1
  || (gr[0][2]==j && gr[1][2]==j && gr[2][2]==j) // colonne 2
  || (gr[0][0]==j && gr[1][1]==j && gr[2][2]==j) // diagonale descendante
  || (gr[0][2]==j && gr[1][1]==j && gr[2][0]==j); // diagonale montante
}
```

3.9 Procédure de jeu (finalisé)

Complétez votre procédure.

Testez. Exemple d'exécution (Fin par une victoire) :

```
Le tour au joueur 2
Coordonnees (x,y)? ( 2 , 0 )

xy 0  1  2
----------
0| x  .  .
1| o  x  .
2| o  .  .

Le tour au joueur 1
Coordonnees (x,y)? ( 2 , 2 )

xy 0  1  2
----------
0| x  .  .
1| o  x  .
```

```
2| o  .  x

Le joueur 1 a gagne
```

Validez votre procédure de jeu avec la solution.

Solution C @[pgmorpion.c]

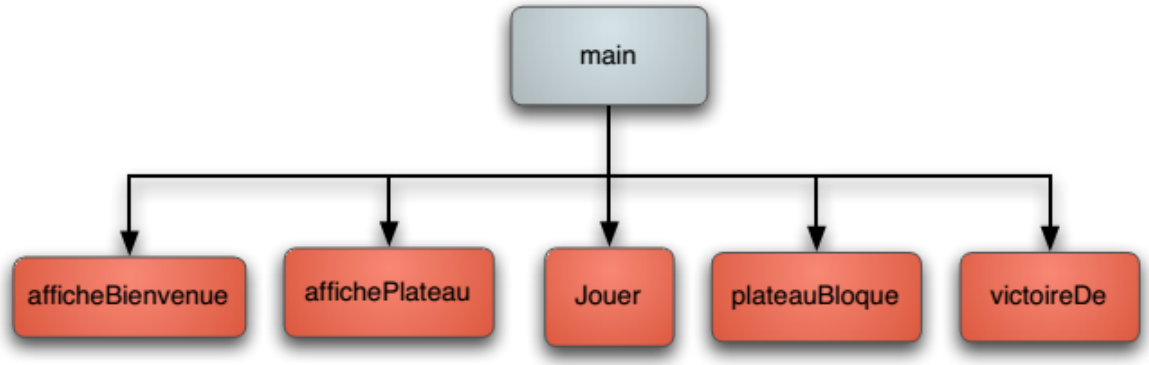
```c
void jeuMorpion()
{
  Grille gr; // plateau du jeu
  int quiJoue = 1; // le prochain a jouer
  initialiserPlateau(gr);
  afficherBienvenue();
  afficherPlateau(gr);
  bool finJeu = false;
  while (!finJeu)
  {
    jouer(quiJoue, gr);
    afficherPlateau(gr);
    if (victoireDe(quiJoue, gr))
    {
      printf("%d gagne\n",quiJoue);
      finJeu = true;
    }
    else if (plateauBloque(gr))
    {
      printf("Jeu bloque\n");
      finJeu = true;
    }
    else
    {
      quiJoue = joueurSuivant(quiJoue);
    }
  }
}
```

Générez la documentation HTML finale de votre programme.

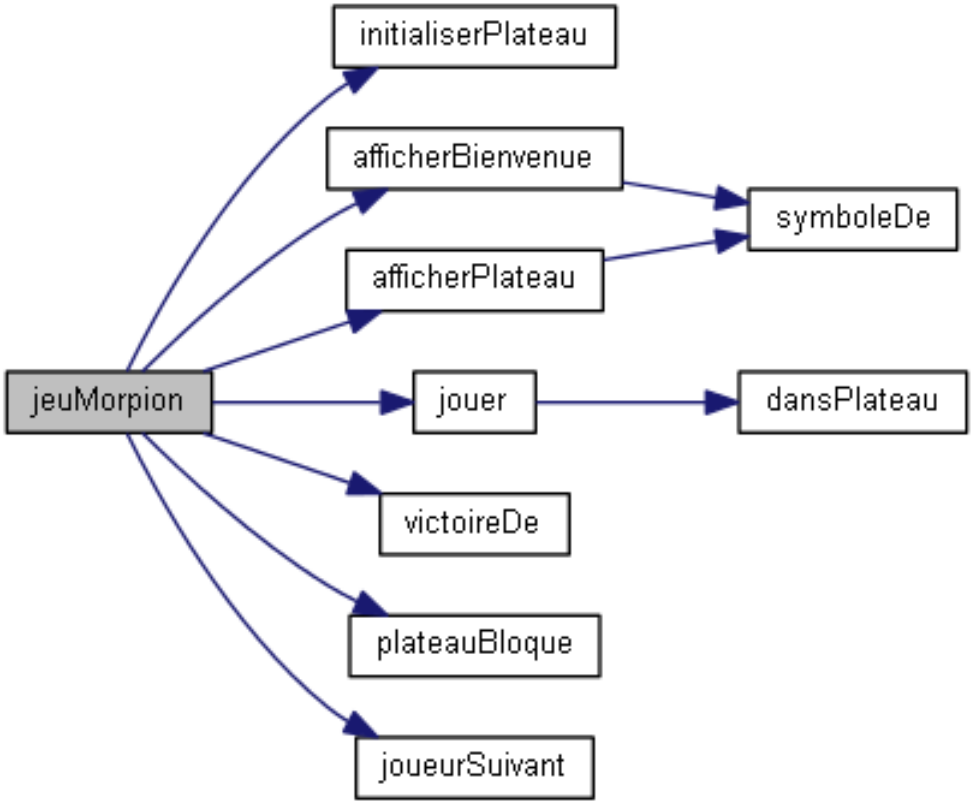
...(suite page suivante)...

3.10 Retour sur l'ébauche

Dans @[Ébauche du programme] de la section @[Analyse], nous avions défini le diagramme suivant :



Dans cette section, nous obtenons le graphe des appels suivant :



Nous constatons que les objectifs ont été réalisés.

4 Robustesse et correction

4.1 Définitions et propriétés

Définition

Un **programme** est :

- **Correct** s'il accomplit la tâche pour laquelle il a été conçu.
- **Robuste** s'il est capable de traiter les entrées invalides ou non attendues de manière raisonnable.

Exemple

Un programme de tri d'entiers saisis au clavier est :

- **Correct** s'il réalise correctement le tri.
- **Robuste** s'il ne plante pas avec des entrées non numériques.

Propriété

Tous les programmes :

- **Doivent être corrects** : un tri qui ne trie pas ses entrées n'est pas utile.
- **Ne doivent pas nécessairement être très robustes** : tout dépend de leur utilisation. Un utilitaire personnel n'a pas besoin d'être très robuste. Un logiciel de la SNCF a intérêt à être très robuste.

Améliorer la qualité des programmes

Il existe des outils et techniques afin de favoriser :

- La robustesse : Utilisation des exceptions (cf. @[Les exceptions])
- La correction : Documentation avec pré-conditions et post-conditions.

4.2 Pré-conditions et post-conditions

Définition

Les **pré-conditions** sont les **conditions préalables** à l'exécution de l'opération afin d'assurer qu'elle s'exécute correctement.

Les **post-conditions** sont les **faits (résultat, comportement) postérieurs** à toute exécution correcte de l'opération.

Intérêts

Elles sont utiles pour la conception et l'évolution des programmes.

Exemple

```
double puissance(double x, int n)
{
  double rs = 1.0;
  int j;
  for (j = n; j >= 1; --j)
  {
    rs = rs*x;
  }
  return rs;
}
```

Explications :

- Pour donner un résultat correct, la fonction a besoin de la pré-condition $n \geq 0$.
- Si la pré-condition est vérifiée, en résultat, elle renvoie la post-condition x^n.

4.3 Contrat d'une opération

Définition

Les pré/post-conditions sont des **contrats** :

- Pré-condition ⇔ **contrat d'entrée** :
  Si le contrat est respectée, l'opération s'exécute correctement.
- Post-condition ⇔ **contrat de sortie** :
  Spécifie ce que « fournit » l'opération si le contrat d'entrée est respecté.

Exemple

```
/**
  Puissance n-ème d'un réel
  @pre n >= 0
  @post x^n
*/
```

Quel est le contrat de l'opération suivante ?

```
Action saisirPositif( R n : Entier )
Début
  |  Répéter
  |     |  Afficher ( "Un entier positif? " )
  |     |  Saisir ( n )
  |  Jusqu'à ( n > 0 )
Fin
```

Solution simple

On a :

- Pré-conditions : Aucune
- Post-conditions : Un entier $n > 0$

Après exécution de cette procédure, nous sommes certain du fait que $n > 0$. Notons que nous aurions pu définir une fonction au lieu d'une procédure.

Utilité des contrats

Les contrats nous indiquent :

- Pré-conditions : ce que nous devons assurer **avant** chaque appel.
- Post-conditions : ce sur quoi nous pouvons compter **après** l'opération (c.-à-d. une fois l'opération exécutée).

Ainsi les post-conditions nous aident à raisonner sur les résultats de notre programme et donc sur sa correction.

4.4 Contrats des opérations au morpion

Répondez aux questions en **complétant** votre programme :

```
/**
  @pre ...
  @post ...
*/
```

Quels sont les contrats pour `victoireDe(j,gr)` ?

Solution simple

Il est clair qu'elle ne fonctionne que si l'on joue sur un plateau de 3×3 :

```
@pre gr est une grille avec des valeurs dans [0..2]
@post Vrai si une ligne, colonne ou diagonale est remplie avec la valeur de j
```

Quels sont les contrats pour `symboleDe(j)` ?

Solution simple

Elle suppose que `j` vaut 0, 1 ou 2 :

```
@pre j dans [0..2]
@post le symbole dans '.','x','o' associé à j
```

Quels sont les contrats pour `afficherPlateau(gr)` ?

Solution simple

Nous devons inclure les pré-conditions nécessaires pour l'appel à `symboleDe` :

```
@pre gr est une grille dont toutes les cases sont dans [0..2]
```

Quels sont les contrats pour `jouer(j,gr)` ?

Solution simple

Nous avons :

```
@pre j doit être compris dans [1..2]
@pre Les cases de gr doivent contenir des entiers dans [0..2]
@pre gr a au moins une case vide (pas plein)
@post gr a une case cochée de plus égale à j (1 ou 2)
```

Refaites la génération HTML de la documentation de votre programme.

...(suite page suivante)...

4.5 Programme à analyser

Soit la procédure `pgmAvecErreur` qui fait jouer les **deux joueurs l'un après l'autre dans la même boucle** :

```
void pgmAvecErreur()
{
  Grille gr; // plateau du jeu
  int quiJoue = 1; // le prochain a jouer
  initialiserPlateau(gr);
  afficherBienvenue();
  afficherPlateau(gr);
  bool finJeu = false;
  while(!finJeu)
  {
    jouer(1, gr);
    afficherPlateau(gr);
    jouer(2, gr);
    afficherPlateau(gr);
    if (victoireDe(1,gr))
    {
      printf("Le joueur 1 gagne\n");
      finJeu = true;
    }
    else if (victoireDe(2,gr))
    {
      printf("Le joueur 2 gagne\n");
      finJeu = true;
    }
    else if (plateauBloque(gr))
    {
      printf("Jeu bloque\n");
      finJeu = true;
    }
  }
}
```

Cette procédure est incorrecte.
Pourquoi ?

Solution simple

Elle est incorrecte car elle :

- Fait jouer les 2 joueurs à la suite sans tester entre les deux :
  - Si l'un des deux a gagné
  - Ou si le plateau s'est rempli
- En fin de boucle, elle teste pour chacun des joueurs s'il a gagné et signale le gagnant (le premier à faire un alignement).

Elle souffre donc de deux petites « anomalies » :

- Elle fait jouer le 2e joueur une fois de trop si auparavant le 1er a joué un coup gagnant.
- Si les deux joueurs font un alignement en même temps, elle dira que le joueur 1 a gagné (alors que le joueur 2 a aussi gagné).

Y a-t-il un autre problème ?

Solution simple

Examinons le contrat de la procédure `jouer` :

```
@pre j doit être compris dans [1..2]
@pre Les cases de gr doivent contenir des entiers dans [0..2]
@pre gr a au moins une case vide (pas plein)
@post gr a une case cochée de plus égale à j (1 ou 2)
```

Que se passe-il lorsque nous exécutons deux fois de suite cette opération ?

```
jouer(1,gr)
afficherPlateau(gr)
jouer(2,gr)
afficherPlateau(gr)
```

- La post-condition de `jouer(1,gr)` nous dit que `gr` a une nouvelle case de cochée.
- Il se peut donc que `gr` soit pleine.
- La pré-condition de `jouer(2,gr)` nous dit que `gr` doit être non plein ==> elle n'est pas respectée !

Dans la procédure `jouer`, que se passe-t-il si la pré-condition « gr est non plein » n'est pas respectée ?

Solution simple

Le joueur ne pourra donner aucune coordonnée, car tout est occupé. S'il en donne, on lui demandera de recommencer lors du test :

```
Si (gr[x,y] <> 0) Alors...
```

La saisie **devient infinie**.

Concluez.

Solution simple

Donc, si la pré-condition n'est pas respectée, l'opération `jouer` reste bloquée en mode « saisie ». Ce n'est pas une erreur facile à faire apparaître : il faut que le plateau soit rempli par le joueur 1 (et pas par le joueur 2).

Finalement quel est l'intérêt des contrats dans ce problème ?

Solution simple

L'étude fine des contrats des opérations nous a permis de comprendre le problème du programme : il ne respecte pas la pré-condition pour le deuxième appel à `jouer`. Pour le résoudre, nous pouvons nous assurer que le test de plateau non plein est fait entre deux coups, ou revenir à la version précédente qui était correcte.

5 Que retenir de cet exercice?

- L'**analyse descendante** permet une mise en oeuvre par étapes d'un programme.
- Les **pré/post-Conditions** renseignent sur le contrat d'utilisation d'une opération mais pour que cela soit utile, il faut s'en servir.

6 Références générales

Comprend [Boudreault-CC1], [Carrez-AL1], [Rohaut-JV1 :c5 :xm] ■