
MASTER 2 - Semestre 1 - 2008

# *Qualité et fiabilité logicielle*
# *Puissance 4 en Java et tests logiciels*

MALEVILLE Nicolas
LAMARTI Abdessamad
DUBERNET Jean Sébastien
CRANSAC Dorian
EWANS Edouard
SELLIER Xavier

Enseignant : Rollet

# Table des matières

# 1 Spécifications

Le programme ici spécifié donne la possibilité à un utilisateur de jouer au jeu 'Puissance 4' dans les conditions décrites ci-dessous :

## 1.1 Application des règles de puissance 4

Règles concernant les joueurs :
Le jeu fait intervenir deux joueurs. Chaque joueur possède des jetons d'une couleur propre et ajoute lors de chaque tour un jeton dans une des colonnes d'une grille de dimensions n x m (par défaut 6 x 7). Un joueur est déclaré gagnant s'il aligne quatre jetons verticalement, horizontalement ou en diagonale dans la grille. Si la grille est remplie alors qu'aucun joueur n'a été déclaré gagnant, la partie est nulle.

Propriétés de la grille :
Pour jouer à Puissance 4, on utilise comme support une grille dont les dimensions standard sont de 6 lignes et 7 colonnes, que l'on numérotera respectivement de 0 à 5 et de 0 à 6. Ce support doit être présent dans notre programme.

Déroulement de la partie :
Au début de la partie, la grille est vide. Pendant la partie, des jetons de 2 couleurs différentes sont ajoutés successivement dans la grille jusqu'à ce qu'elle soit remplie ou que l'un des joueurs forme un alignement de 4 jetons de sa couleur. Un jeton est ajouté à la grille en fournissant le numéro de colonne. En effet, le numéro de ligne ne fait pas partie du choix du joueur dans le sens où un jeton descend nécessairement au niveau le plus bas accessible dans la grille. Par exemple, le premier jeton descend à la ligne numero 5 (la plus basse). Si une colonne est remplie (6 jetons on été insérés dans la même colonne), aucun joueur ne peut jouer de jeton dans cette colonne jusqu'à la fin de la partie.

On s'assurera que chacune de ces règles sera respectée dans notre programme.

## 1.2 Configurations de jeu et types de joueurs

Un joueur peut être un humain ou une entité controllée par l'ordinateur. Ainsi, on peut avoir les configurations de jeu suivantes : cpu vs joueur ou joueur vs joueur. Quelle que soit la configuration, le programme contiendra une entité représentant le joueur 1 et le joueur 2. En fonction du mode choisi par l'utilisateur, le joueur 2 sera humain ou controllé par l'ordinateur. Le joueur 1 sera toujours humain. Un humain commence toujours la partie.

- IA : tous les deux tours la main est donnée à l'ordinateur. On veut pouvoir choisir différents niveaux de difficulté (i.e : l'ordinateur est plus ou moins performant selon le niveau choisi). Il doit être possible d'ajouter de nouveaux modes de difficulté facilement. Chaque

niveau sera représenté par un algorithme qui déterminera le prochain coup que l'ordinateur va jouer (i.e le prochain numéro de colonne).

- humain : tous les deux tours la main est donnée à l'utilisateur. A chaque fois que le tour d'un humain vient, le programme attendra que l'utilisateur saisisse son choix.

Quelque soit le type de joueur, un choix de colonne doit être effectué à chaque tour de manière à ce que le programme s'exécute correctement. Dans le cas du joueur humain, on attendra que l'utilisateur fasse son choix grâce à une interaction avec l'interface (graphique ou dans le terminal).

### 1.3 Visualisation

L'utilisateur du programme doit pouvoir visualiser en temps réel le déroulement du jeu, c'est-à-dire les coups joués par l'ordinateur ou l'humain. Dans un premier temps, on utilisera un affichage en mode console, puis si le planning le permet, une interface graphique plus agréable pour l'utilisateur. L'outil de visualisation doit être facilement interchangeable.
En premier lieu, l'utilisateur doit pourvoir choisir, au travers de l'interface, quel type de configuration il souhaite utiliser et instancier une partie.
Ensuite, lors du déroulement de la partie, il doit pouvoir connaître grâce à l'affichage quels sont les coups qui ont été joués jusqu'à l'instant où il lui est demandé de jouer. Enfin, il doit pouvoir interagir avec le panneau visuel de manière à faire connaître son choix pour le coup en cours au programme lorsque c'est à lui de jouer. Lorsqu'une partie est terminée, l'utilisateur doit être averti de l'issue de celle-ci (joueur 1 gagne, joueur 2 gagne ou match nul). Eventuellement, il doit alors pouvoir jouer une nouvelle partie.

### 1.4 Tests associés

# 2 Architecture

Nous allons décrire ici les différents modules imaginés pour répondre aux attentes formulées dans la partie spécification.

## 2.1 Structure de données

Nous implémenterons une classe contenant un tableau d'entiers et les différentes méthodes nécessaires aux accès en écriture et lecture sur celui-ci. Ainsi, on doit pouvoir instancier un tableau de dimensions variables, initialiser ses valeurs et les modifier. Le tableau doit également pouvoir être réinitialisé. Cette classe a pour but de représenter la grille et donc les coups joués par chaque joueur au long de la partie. On représentera une case vide par un entier nul à la position correspondante. Un entier dont la valeur est à 1 représentera un coup joué par le joueur 1, même chose pour le joueur 2.
C'est ce module qui va permettre au joueur ordinateur de connaître les possibilités de jeu à chaque tour (coups jouables). C'est grâce à cette représentation également que l'on saura si la partie est terminée ou non. Enfin, l'interface sera mise à jour à chaque tour en fonction de ce tableau, ce qui évitera des possibilités de confusion entre les deux entités.

## 2.2 Joueurs

Il sera nécessaire d'implémenter 2 classes différentes pour les 2 types de joueurs. Cependant, certaines méthodes ou variables étant communes à ces entités on prévoiera une interface regroupant les méthodes nécessaires au jeu et appelées quelque soit le type de joueur. Ceci permettra au module en charge du déroulement du jeu de faire abstraction de la configuration choisie, et donc de diminuer le volume de code.

### 2.2.1 Humain

Le joueur humain aura une connexion étroite avec l'interface choisie. En effet, jouer un coup reviendra à interroger l'utilisateur via cette interface. Cette classe sera donc relativement peu volumineuse et d'une complexité faible.

### 2.2.2 Ordinateur

L'entité représentant le joueur artificiel sera plus importante. On devra pourvoir choisir entre 2 niveaux de difficultés, ce qui se traduira par une instanciation selon un paramètre particulier, comme un entier ou un boolén. Ainsi, la classe représentant ce type de joueur contiendra des méthodes différentes et correspondant chacune à ce niveau et donc paramètre choisi. On choisira premièrement un algorithme simple représentant une stratégie triviale (exemple : jouer la même colonne ou une colonne voisine de celle choisie par le joueur 1 précédemment), puis on cherchera à implémenter un algorithme plus 'intelligent' permettant au joueur humain de se confronter à une difficulté de jeu élevée. On se rapprochera d'une stratégie gagnante pour le joueur-ordinateur.

### 2.3 Règles

Nous avons choisi d'implémenter un module destiné à l'application des règles tout au long de la partie. De manière plus spécifique, nous nous intéressons ici à la detection d'une situation incorrecte, d'une situation de terminaison ou bien du simple déroulement d'un coup joué.

- Situation incorrecte
  La principale situation incorrecte dans notre système correspond au cas où un joueur tente d'ajouter un jeton dans une colonne déjà pleine. Le module en question contiendra les méthodes nécessaires à la detection de ce type de coup, méthodes que l'on utilisera alors pour avertir l'utilisateur de cette interdiction.

- Situation de terminaison
  Une situation de terminaison se produit dans deux cas :
  Premièrement, un joueur aligne 4 pions dans la grille.
  Deuxièmement, la grille est remplie.
  De la même manière, on souhaitera que ces évènements soient détectés et traités en conséquence. C'est à dire, information de l'utilisateur et terminaison éventuelle du programme.

- Déroulement d'un coup joué
  Lorsqu'un joueur communique son choix pour le coup en cours, le tableau d'entiers doit être modifié en conséquence. Nous avons décidé d'affecter cette tâche à ce module, qui sera donc en étroite corrélation avec la structure de données. Pour effectuer cette modification, la méthode correspondante aura connaissance du numéro du joueur en cours et la colonne choisie par ce joueur. L'action sera alors traduite par une affectation en accord avec les méthodes de la structure de données.

L'ensemble de ces outils permettront au moteur de jeu décrit ci-dessous d'assurer le bon déroulement du jeu et donc du programme.

### 2.4 Moteur de jeu

Le moteur de jeu sera le coeur de notre programme dans le sens où il mettra en rapport l'ensemble des entités du programme. Ainsi, il devra assurer le scénario standard suivant :

1. Phase d'initialisation
2. Phase de jeu (a)
3. Phase de modification (b)
4. Phase de communication (c)
5. Terminaison ou nouvelle partie

La phase d'initialisation correspond à la création et au lancement de chacun des modules cités dans cette section : une grille de dimension choisie sera initialisée, les joueurs seront instanciés conformément au choix initial de l'utilisateur, l'interface utilisateur sera lancée au besoin, une instance du module de règles sera créée.

La phase de jeu correspond à l'ordonnancement des coups des joueurs. Tour à tour, le moteur de jeu consultera le joueur concerné et lui demandera de fournir le numéro de colonne qu'il souhaite jouer, qu'il soit humain ou ordinateur.

La phase de modification permettra de traduire ce choix en une consultation du module de règles et modification de la grille en conséquence.

La phase de communication représentera la notification au joueur via l'interface de la configuration actuelle du jeu. Ainsi, dans le cas d'une interface graphique on mettra à jour l'affichage du panneau en fonction du nouvel état de la grille.

Ces trois dernières phases (a-b-c) seront répétées jusqu'à ce qu'une situation de terminaison apparaîsse.

Enfin, l'utilisateur choisira de terminer le programme ou de jouer une nouvelle partie. Le module de jeu sera donc capable de se réinitialiser pour revenir à une configuration initiale.

Chacune de ces phases sera représentée par une ou des méthodes contenue dans une classe dédiée.

### 2.5 Composants annexes

On trouvera d'autres classes d'importance mineure :

Une classe main sera présente et permettra d'instancier le moteur de jeu au démarrage du programme en lui communiquant le choix de configuration de l'utilisateur. D'autres interfaces pourront être ajoutées pour augmenter la modularité du programme.
Enfin, dans le cas d'une interface graphique, il sera nécessaire de représenter chaque position de la grille grâce à des éléments graphiques changeant de couleur. De la même manière, d'autres composants graphiques permetront à l'utilisateur de communiquer son choix quant au coup qu'il veut jouer. On trouvera également une zone graphique destinée à l'avertir de la terminaison de la partie.

# 3 Implementation

## 3.1 DataStructure

### 3.1.1 Matrice

```
private int[][] matrix;
private int height;
private int width;
```

Nous avons choisi d'utiliser une matrice pour modéliser une grille de Puissance 4. Ce sera un tableau d'entiers. Afin de faire moins de calculs nous avons choisi de stocker la hauteur ainsi que la largeur de la matrice dans deux variables distinctes `height` respectivement `width`.

```
public DataStructure(int height, int width);
```

Le constructeur va prendre deux entiers en paramètres. Ce constructeur va vérifier si ces entiers sont valides, autrement dit si ils ne sont pas négatifs ou nuls.

```
public int getHeight();
public int getWidth();
public int getValue(int i, int j);
public boolean setValue(int i, int j, int color);
```

Nous avons préféré mettre nos variables (`matrix`, `height` et `width`) en private pour éviter toutes modifications inattendues de notre matrice. d'où l'existence de ces accesseurs. Nous n'avons pas fait d'accesseur direct à la matrice, autrement dit qui retournerait notre matrice, toujours dans le soucis de modifications inattendues.

La méthode `setValue` va modifier notre matrice en respectant la contrainte disant que `i` et `j` doivent etre compris entre 0 et respectivement `height` et `width`.

Nous n'avons pas mis de restriction sur color, etant donné que la structure de données ne gère pas les règles du jeu, elle ne sait pas de quoi il s'agit exactement.

Cette méthode retourne un booléen qui renvoie `true` respectivement `false` si la modification a pu être apporté ou pas.

```
public void reset_matrix();
public void print();
```

La méthode `reset_matrix` va, comme son nom l'indique, faire un simple reset de la matrice.

La méthode `print` va elle afficher la matrice.

### 3.1.2 Tests de la matrice

Nos différentes classes de test dérivent de **junit.framework.TestCase**. Elles ont une méthode setUp(), exécutée avant chaque méthode de test, pour initialiser les tests et une méthode tearDown(), exécutée après chaque méthode de test , pour relâcher les données.
Les méthodes de test n'attendent pas de paramètres et retournent **void**. Leur nom commence par **testXXX**.
A l'intérieur des méthodes de test, on effectue les actions de test souhaitées, et on vérifie

qu'elles se passent correctement en utilisant des méthodes **assertXXX**.

Ces tests consistent à s'assurer que la structure de données, qui représente notre grille de Puissance 4, est robuste.

Le premier test envisagé est d'instancier une `DataStructure` 6×7 correspondant à la grille classique. Tout en vérifiant par les accesseurs que les dimensions de la matrice sont celles attendues.

On réalise ensuite des tests aux limites sur notre structure de données, par exemple 0×0. Pour continuer sur quelques tests aléatoires 100×100.

Nous allons tester des ajouts de valeurs prises de manière aléatoire mais qui restent valides. Et vérifier le comportement de `setValue` aux bornes. Donc en valeur `i` et `j` négatives ou supérieures à `height` et à `width`.

Exemple :
**testAddingMoreThan42Values()** :
Mettre une valeur à une position de la grille s'écrit : **matrix.setValue(i, j, c)**. Cette action doit retourner un booléen **true**, ce qui est testé par l'instruction :

**assertTrue(matrix.setValue(i, j, c)) ;**

Une fois toutes les cases de la grille remplies, le test d'ajout ne doit plus passer, ce qui s'écrit :

**assertFalse(matrix.setValue(i, j, c)) ;**

Un **test aux limitex** a été mis en place dans cette méthode : En plus de tester la possibilité que toutes les cases de la grille soient remplies, nous avons testé le rajout de valeurs à des positions *hors limite*.
Exemple :
**assertFalse("Test aux limites : limite inf", matrix.setValue(-1, -1, 1)) ;**
**assertFalse("Test aux limites : limite sup", matrix.setValue(6, 0, 0)) ;**

## 3.2 Player

`Player` est une interface permettant de basculer facilement entre un `HumanPlayer` et un `CpuPlayer`. Nous reverrons cette partie dans le `GameEngine`.

### 3.2.1 HumanPlayer

```
private int currently_played;
```

Sert a mémoriser la position que le joueur a choisi.

```
public int play(DataStructure grid, GUI gui);
```

Cette méthode va attendre que le joueur face son choix à travers l'interface graphique. Elle récupère aussi bien une position jouée qu'un reset de la grille. Cette méthode retourne la position choisit par le joueur mais ne vérifie pas si elle est correcte. Cela s'effectue plus loin au moment du `GameEngine`.

### 3.2.2 CpuPlayer

```
private int mode;
private int currently_played;
private Rules rule;
```

La première variable va stocker le mode, 0 pour deux humains, 1 pour un ordinateur facile et 2 pour un ordinateur difficile.

La seconde variable, `currently_played`, stocke la position jouée par le joueur (humain ou ordinateur).

La troisième variable, `rule`, va stocker les règles du jeu. Cela est utile pour l'ordinateur, afin de bien respecter les règles du jeu lorsqu'il effectue un choix. Nous reverrons plus en détail cette partie dans la classe `IaFourInARow` qui implémente l'intelligence artificielle.

```
public CpuPlayer(int mode, Rules rule);
public int play(DataStructure grid, GUI gui);
```

Notre première méthode est un constructeur. Elle attribut donc un `mode` au CpuPlayer mais aussi les règles a utiliser (`Rules`).

Notre seconde méthode, `play`, similaire à celle de HumanPlayer. Elle va quand a elle vraiment instancier l'intelligence artificielle en fonction du mode choisit. Cela se fait à l'aide des lignes 21-22 :

```
Cpu cpu1 = new IaFourInARow();
cpu1.initialize(grid, mode);
```

Si on veut utiliser une autre intelligence artificielle, il suffirait de créer une nouvelle classe qui implémente `Cpu` et qui fonctionne globalement comme l'IA actuelle. Notre Intelligence artificielle est donc totallement indépendante du reste du programme et peut être changé facilement.

### 3.2.3 Jeux de test

## 3.3 Cpu

`Cpu` est une interface. le fait d'avoir créé une interface va nous permettre de pouvoir implémenter de nouvelles IA sans modifier plus d'une ligne du code actuel.

### 3.3.1 IaFourInARow

Pour ce qui concerne l'intelligence artificielle, nous avons choisi d'en implémenter une séquentielle. Il n'y aucune part d'aléatoire. Cette IA n'implémente pas la stratégie gagnante. Nous nous sommes renseigné et il faudrait implémenté un arbre faisant intervenir des statistiques de meilleurs coups a jouer. cela ne semble pas évident au premier abord à implémenter, c'est pourquoi nous avons implémenté une IA maison qui donne des résultats assez satisfaisant car non triviale.

cependant si on veut implémentr la stratégie gagnante cela est possible en créant une classe qui implémente l'interface `Cpu` et qui respecte le fonctionnement de l'IA actuelle qui est décrit ci-après.

```
private int mode;
private DataStructure cpugrid;
private int[] playable;
```

Comme toujours le `mode` stocke la difficulté de l'IA, 1 pour facile, 2 pour difficile.

`cpugrid` n'est pas une copie de la grille, mais un pointeur sur la grille, donc si on modifie cpugrid on modifie aussi la grille de puissance 4. Nous avons fait ce choix pour éviter de recopier la grille à chaque fois, mais cela ajoute de la contrainte de ne pas modifier la grille lors du calcul de notre stratégie.

La variable `playable` va etre de la taille de `width`. Elle va servir pour déterminer quelle colonne jouer ou quelle colonne ne pas jouer. tout cela se fait à l'aide d'un codage simple.

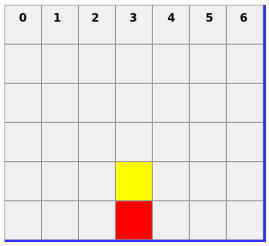


FIG. 1 – `playable[i] = 0`

Dans ce cas, tous les playable[i] sont égaux à 0. autrement dit on peut jouer sur n'importe quelle case. Seulement notre intelligence artificielle choisiera de jouer au centre du jeu, car la probabilité de gagner lors qu'on joue au centre est supérieure aux autres (par exemple si on joue sur les cotés).

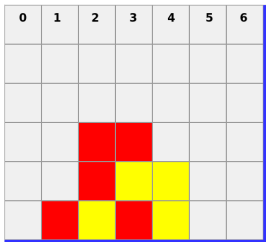


FIG. 2 – `playable[i] = 1`

Si l'ordinateur joue sur la colonne 4, alors au prochain coups le joueur humain pourra gagner avec une diagonale. Par conséquent `playable[4]=1`. Ce code signifie que si l'ordinateur joue sur cette colonne alors cela peut faire gagner le joueur humain.

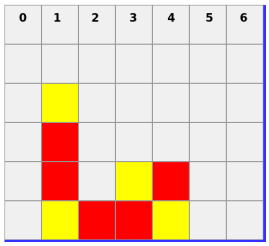


FIG. 3 – `playable[i] = 2`

Dans ce cas `playable[2] = 2`. Autrement dit, si l'ordinateur joue sur la colonne deux, le joueur humain pourra le bloquer.

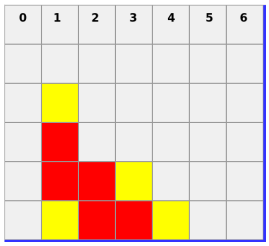


FIG. 4 – `playable[i] = 3`

Dans ce cas `playable[2] = 3`. Par conséquent l'ordinateur peut gagner ua prochain coups, et jouera la position 2.

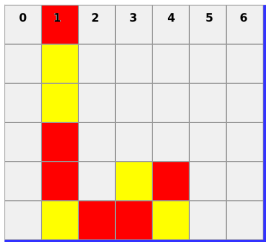


FIG. 5 – `playable[i] = 4`

La colonne 1 est pleine donc `playable[1] = 4`. L'ordinateur prend conscience qu'il ne pourra pas placer de jetons dans cette colonne.

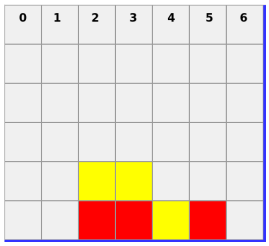


FIG. 6 – `playable[i] = 5`

A ce moment du jeu, l'ordinateur remarque qu'il peut gagner en ajoutant 2 pions. Par conséquent il va marquer `playable[4] = 5` et `playable[5] = 5`.

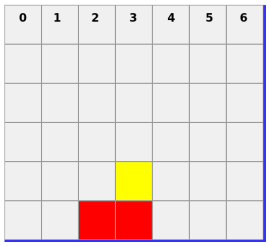


FIG. 7 – `playable[i] = 6`

Dans ce cas la `playable[1] = 6` et `playable[4] = 6`. Et tous les autres cas `playable[i]=0`. L'ordinateur va bloquer la possibilité de mouvement de l'humain.

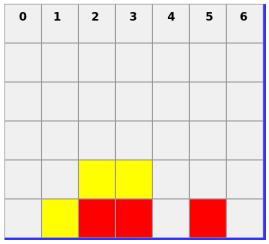


FIG. 8 – `playable[i] = 6`

Si le joueur humain place un pion dans la colonne 4 il remportera la victoire. Pour eviter de perdre aussi facilement on met `playable[4]=6`, et l'ordinateur va bloquer la victoire du joeur humain.

```
private Rules rule;
private int height;
private int width;
```

La première vriable va contenir les règles du jeux qui sont utilisées, dans notre cas ce sera pour le Puissance 4. Les deux variables suivantes contiennent la hauteur, respectivement la largeur de notre `cpugrid`.

```
private void strategy();
private void breakStrategy();
private void noPlayable();
private void winningPlayable();
private void fillPlayable();
```

La méthode `strategie()` va parcourir la grille et calculer les positions que l'ordinateur peut jouer pour pouvoir gagner en deux coups. une fois qu'elle a repérer les colonnes et remplie en conséquence `playable[i]`. La méthode suivante, `breakStrategy()` va parcourir la grille et va regarder si lorsque l'ordinateur joue un coup, alors cela permettra a l'humain de la bloquer alors qu'avant il ne le pouvait pas. (pour plus de détails voir les captures d'écran précédemment). Dans ce cas `playable[i] = 5`.

La méthode `noPlayable` a un double usage. tout d'abord elle regarde si le joueur humain peut gagner au prochain coups. Dans ce cas elle marque `playable[i] = 6` (i correspondant à la colonne). Mais elle va aussi vérifier si en jouant une colonne cela permet à l'humain, au coups d'après, de gagner la partie. Dans ce cas `playable[i] = 1`.

Pour `winningPlayable()`, on regarde si l'ordinateur peut gagner au prochain coups, et on marque la colonne correspondante par 3 autrement dit `playable[i] = 3`.

La méthode `fillPlayable()` quand a elle initialise `playable[i] = 0` pour les colonnes ou on peut encore jouer. Pour ce qui concerne les colonnes pleines, alors `fillPlayable()` va mettre `playable[i]=4`.

```
public void initialize(DataStructure grid, int difficulty);
public int play(Rules new_rule);
```

La méthode `initialize()` nést pas un constructeur, mais comme son nom l'indique, elle va initialiser le `Cpu`. Quand à la méthode `play()` elle va vérifier selon le `mode`, autrement dit la `difficulty`, quelle IA appeler.

```
public int perfectCpu();
public int easyCpu();
```

Ces deux méthodes sont très similaires, à la différence que le `perfectCpu()` appelle les méthodes `breakStrategy()` et `strategy()`, ce que ne fais pas la méthode `easyCpu()`. Le mode `easyCpu` est moins agressif que le mode `perfectCpu()`, qui en passant n'est pas parfait ;-)
Dans notre méthode `perfectCpu()` on commence par remplir la grille avec `fillGrid()` et continue par vérifier si on peut gagner au prochain coup à l'aide de la méthode `winningPlayable()`. Si il y a une position gagnante alors on retourne le numéro de la colonne. Sinon on regarde si le joueur humain peut gagner au prochain coups, ou si jouer une certaine colonne peut le faire gagner. Tout cela à l'aide de la méthode `noPlayable()`.

Si on est dans le `perfectCpu()` alors on établie une stratégie. Pour cela on évite de jouer une colonne ou l'humain pourrait nous bloquer par la suite, avec la méthode `breakStrategy()`. Ensuite on essaie d'établir une stratégie, si on a 2 jetons alignés, alors on va essayer dén aligner 2 de plus pour faire un 4 à la suite, à l'aide de la méthode `strategy()`. Cela ne se fait que dans la méthode `perfectCpu()`

On continue, peu importe la méthode `perfectCpu()` ou `easyCpu()`, on regarde si on a autre chose que des 0, 1 et 2 dans `playable[i]`, et dans ce cas on remplit les colonnes du milieu de la grille. Sinon dans l'ordre des priorités pour `playable[i]` l'ordinateur joue de cette manière : `3 - 6 - 5 - 0 - 2 - 1`.

### 3.3.2 Jeux de test

TODO

## 3.4 Rules

`Rules` est une interface. Cela nous permet d'implémenter de nouvelles règles. Ca peut être utile si on veut transformer notre Puissance 4 en Morpion, ou si on veut faire un Puissance 5 ...

### 3.4.1 FourInARow

Cette classe implémente `Rules`.

```
public boolean checkDiag(int i, int j, int color, DataStructure grid);
public boolean checkCol(int i, int j, int color, DataStructure grid);
public boolean checkLine(int i, int j, int color, DataStructure grid);
public boolean isComplete(DataStructure grid);
public boolean checkPlay(int play, DataStructure grid);
public void greyOut(GUI app, DataStructure grid);
```

`checkDiag()` retourne `true` si il existe un alignement de 4 jetons d'une même couleur en diagonale. Il retourne `false` sinon. `checkCol()` retourne `true` si il existe un alignement de 4 jetons d'une même couleur en colonne. Il retourne `false` sinon. `checkLine()` retourne `true` si il existe un alignement de 4 jetons d'une même couleur en Ligne. Il retourne `false` sinon. `isComplete()` va faire appel aux trois méthodes précédentes afin de vérifier si il y a un gagnant. Auquel cas cette méthode retourne `true`, sinon elle renvoie `false`. La méthode `checkPlay()` vérifie que la position passée en argument est jouable, elle renvoie `true` si cést jouable, et `false` sinon.

La dernière méthode, `greyOut()`, va quand a elle griser les bouttons des colonnes pleines correspondantes.

### 3.4.2 Tests de FourInARow

Cette classe teste :
L'existence ou non d'un alignement de 4 jetons d'une même couleur soit dans la même ligne, même colonne, même diagonale :
On choisit des valeurs aléatoires valides (constituant une : ligne, colonne, diagonale) afin de vérifier le comportement de nos méthodes d'alignement des jetons.
Exemple (Existence de 4 jetons sur la même ligne) :
On crée 4 jetons sur la même ligne 0 :

**assertTrue(matrix.setValue(0, 0, 1)) ;**
**assertTrue(matrix.setValue(0, 1, 1)) ;**
**assertTrue(matrix.setValue(0, 2, 1)) ;**
**assertTrue(matrix.setValue(0, 3, 1)) ;**

Puis en bouclant sur toutes les positions de la grille, on arrive à effectuer un test complet sur l'existence ou non d'un alignement de 4 jetons sur une ligne donnée.
Qu'une position donnée est jouable en remplissant la grille aléatoirement mais de manière à ce qu'une position soie jouable/non jouable.

## 3.5 GameEngine

### 3.5.1 Implémentation

```
private DataStructure grid;
private boolean current_player;
```

```
private int currently_played;
private GUI app;
private int mode;
private Rules rule;
private Player player1;
private Player player2;
private int counter;
```

La variable `grid` contient la grille du jeu. `current_player` peut obtenir deux valeurs, 0 qui correspond au joueur 1, et 1 qui correspond au joueur 2. `currently_played` contient le coup joué (une position ou un reset). `app` contient l'interface graphique. `mode` contient le mode de jeu. `rule` contient les règles du jeu. `playerX` contient le joueur, qui peut etre humain ou artificiel. `counter` contient le nombre de coups joué, cela nous permet de déterminer si il y a un match nul.

```
public GameEngine();
public void initMode(int my_mode);
public void close();
public void start();
public void resetGrid();
public void updatePlay();

private void updateGrid();
```

Le constructeur initialise l'interface graphique, la grille (avec une taille que l'on peut modifier) et les règles du jeu. La méthode `initMode()` initialise les joueurs, la variable `my_mode` et reset le `counter`. La méthode `close()` quand à elle ferme l'interface graphique. Cette méthode n'est appelée que par le `Main()`.

La méthode `start()` s'occupe de lancer le jeu. il aura été initialisé par les autres méthodes qui la précède.
La boucle `while(( !rule.isComplete(grid)) && (counter < grid.getWidth() * grid.getHeight()))` va faire en sorte que le jeu ne se termine pas tant qu'il n'y a pas de gagnant ou que la grille n'est pas pleine. Il faut savoir que l'utilisateur peut arrêter le programme a tout moment a l'aide de l'interface graphique.

Cette méthode fiat jouer a tour de rôle le joueur 1 et le joueur 2. C'est la méthode `start()` qui va interpréter le choix du joueur, autrement dit, si `currently_played = -2` ca implique qu'il faut reset la grille.

A chaque tour de boucle on incrémente la variable `counter` de un si la position jouée est valide. Par la suite on met à jour la grille à l'aide de la méthode `updatePlay()` et on grise les bouttons dont les colonnes sont pleines à l'aide de la méthode `rule.greyOut()`.

Une fois la boucle terminée, autrement dit, lorsque le jeu est terminé on vérifie qui a gagné et on l'affiche à l'aide de la méthode `gameEnded()`.

La méthode `resetGrid()` s'occupe d'initialiser la `grid` avec des 0, elle fait en sorte de ne aps re-faire appelle à elle avec la méthode `app.setReset(false)` et réinitialise tous les boutons avec la méthode `app.enableAllButton()`. Il faut aussi réinitialiser le `counter` et mettre à jour l'affichage graphique.

La méthode `updatePlay()`, va quand à elle, re-vérifier si la position jouée est valide puis va mettre à jour la grille, et enfin met à jour l'affichage.

La méthode `updateGrid()` va mettre à jour la grille, autrement cést elle qu iva générer la gravité, si on peut dire.

### 3.5.2 Tests du GameEngine

TODO

## 3.6 Main et Menu

Le `Main()` va instancier le `GameEngine` mais aussi le menu de départ. cést le `Main()` qui va dire au `Gameengine` de démarrer le jeu avec l'appel à la méthode `g.start()`. Le `Main()` fait au `GameEngine` instancier les joueurs avec l'appel à la méthode `g.initMode(my_menu.choice)`.

Pour ce qui concerne le menu, on aurait pu l'intégrer dans l'interface graphique, mais nous n'avions pas assez de temps pour faire cette petite modification, et nous avons préféré passer aux tests directement.

## 3.7 GUI

`GUI` est une interface, cela nous permettra, au besoin, de branher une autre interface graphique à notre application.

### 3.7.1 GuiOwn

Cette classe, implémente `GUI`.

```
public int choice;
public boolean played;
public boolean reset;
public boolean game_ended;
```

La première variable `choice` contient la colonne sélectionné par le joueur humain. La seconde variable `played` est a `true` lorsqu'un joueur a choisi une colonne et est a `false` sinon. Il en est de même pour `reset` et pour `game_ended`. A savoir que `reset` concerne le reset de la grille et `game_ended` dit si le jeu est terminé ou non.

```
public abstract void initGui(DataStructure grid);
```

Cette méthode initialise l'interface grapique en créant autant de colonne et de ligne que la matrice en a.

```
public abstract void updateScreen(DataStructure my_grid);
```

`updateScreen` met a jour l'affichage de la grille.

```
public abstract void gameEnded(boolean winner);
public abstract void gameEnded();
```

La première méthode affichage le nom du joueur dans l'interface graphique, et la seconde affiche Match nul.

```
public abstract void greyAllButton();
```

Cette méthode grise tous les bouttons qui permettent de choisir les colonnes mais aussi de faire un reset de la grille. Elle est utilisé lorsque le jeu est terminé.

```
public abstract void enableAllButton();
```

Cette méthode rend tous les bouttons correspondant aux colonnes non-grisés. Elle est utilisé lors d'un reset.

```
public abstract void greyButton(int num);
```

`greyButton()` grise le boutton de la colonne correspondante à num.

```
public void setSize(int i, int j);
public void setLocation(int i, int j);
public void show();
public abstract void dispose();
```

La première méthode permet de définir la taille de la fenêtre de jeu. La seconde sert à définir la position de la fenêtre par défaut. La troisième est une méthode définit dans les librairies de `Swing` qui permet d'afficher l'interface graphique. Quand à la dernière méthode, elle permet de d'afficher/fermer une fenêtre. La méthode `dispose()` est aussi définit dans els librairies de `Swing`.

```
public boolean getPlayed();
public int getChoice();
public void setPlayed(boolean played);
public boolean getReset();
public void setReset(boolean reset);
```

Ces trois méthodes sont de simples accesseurs aux variables correspondantes.

# 4 Analyse Statique