
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 | Introduction | 3 |
| 2 | Spécifications | 3 |
| 2.1 | Application des règles de puissance 4 | 3 |
| 2.2 | Configurations de jeu et types de joueurs | 4 |
| 2.3 | Visualisation | 4 |
| 2.4 | Tests de validation | 4 |
| 2.4.1 | Règles de puissance 4 | 4 |
| 2.4.2 | Configurations de jeu | 5 |
| 2.4.3 | Types de joueurs | 5 |
| 2.4.4 | Visualisation | 6 |
| 3 | Architecture | 7 |
| 3.1 | Structure de données | 7 |
| 3.2 | Joueurs | 7 |
| 3.2.1 | Humain | 7 |
| 3.2.2 | Ordinateur | 7 |
| 3.3 | Règles | 8 |
| 3.4 | Moteur de jeu | 8 |
| 3.5 | Composants annexes | 9 |
| 3.6 | Tests d'intégration | 9 |
| 4 | Implementation | 13 |
| 4.1 | DataSet | 13 |
| 4.1.1 | Matrice | 13 |
| 4.1.2 | Tests de la matrice | 13 |
| 4.2 | Player | 14 |
| 4.2.1 | HumanPlayer | 14 |
| 4.2.2 | CpuPlayer | 15 |
| 4.2.3 | Jeux de test | 15 |
| 4.3 | Cpu | 15 |
| 4.3.1 | IaFourInARow | 16 |
| 4.3.2 | Jeux de test | 20 |
| 4.4 | Rules | 26 |
| 4.4.1 | FourInARow | 26 |
| 4.4.2 | Tests de FourInARow | 26 |
| 4.5 | GameEngine | 27 |
| 4.5.1 | Implémentation | 27 |
| 4.5.2 | Tests du GameEngine | 28 |
| 4.6 | Main et Menu | 29 |
| 4.7 | GUI | 29 |
| 4.7.1 | GuiOwn | 29 |
| 5 | Analyse Statique | 31 |

| | | |
|----------|---------------------------------|-----------|
| 6 | Résultats | 32 |
| 6.1 | Tests de validation | 32 |
| 6.1.1 | Règles de puissance 4 | 32 |
| 6.1.2 | Configurations de jeu | 33 |
| 6.1.3 | Types de joueurs | 33 |
| 6.2 | Tests d'intégration | 34 |
| 6.3 | Tests Unitaires | 36 |
| 6.3.1 | DataStructureTest | 36 |
| 6.3.2 | IAfourInARow | 36 |
| 7 | conclusion | 38 |

1 Introduction

La démarche que nous nous sommes efforcés d'adopter dans ce projet est la suivante. Nous avons suivi le processus du cycle en "V" en concevant les différents niveaux de tests au fur et à mesure de la conception du programme. Nous avons essayé de parcourir ce schéma horizontalement autant que possible, à savoir qu'avant de descendre à un niveau de détail plus élevé, nous nous sommes assurés de concevoir les tests associés au niveau en cours. Le projet étant un exercice scolaire et non pas une application industrielle, les tests du haut du cycle en fin sont plus délicats à mener. Il n'y a pas de "client". Le rapport suit également cette démarche du cycle en "V". Ainsi, chaque partie de la conception sera suivie des tests correspondant à leur contenu.

2 Spécifications

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

2.1 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 $n \times m$ (par défaut 6×7). Un joueur est déclaré gagnant s'il aligne quatre jetons consécutifs 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 numérottera 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 ont 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.

2.2 Configurations de jeu et types de joueurs

Un joueur peut être un humain ou une entité contrôlée 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 contrôlé 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.

2.3 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 pouvoir 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.

2.4 Tests de validation

Nous allons voir ici les tests imaginés pour s'assurer que le programme obtenu respectera les termes précisés dans la partie précédente, et ne générera pas de comportements interdits.

2.4.1 Règles de puissance 4

Voici les tests système liés aux règles du jeu :

– **Test 1**

On lance toutes les configurations disponibles dans le menu et on vérifie que deux entités peuvent jouer des coups de couleur différente.

– **Test 2**

On instancie une grille de taille 6x7 puis de tailles différentes.

– **Test 3**

On utilise le mode humain vs humain pour arriver à une partie nulle et on vérifie que l'on est averti de cette issue.

On soumettra ensuite le programme à un échantillon d'utilisateurs connaissant les règles et on recueillera leur avis concernant leur application.

– **Test 4**

Demander aux personnes composant l'échantillon si elles jugent que les règles de puissance 4 attendues ont été respectées pendant leur partie.

Enfin, on pourra utiliser le joueur ordinateur pour générer des parties aléatoires et en vérifiant à chaque fois que l'empilement des pions respecte les règles.

– **Test 5**

Générer des parties aléatoires et vérifier les coups joués.

2.4.2 Configurations de jeu

Il suffit d'un test fonctionnel basique dont l'objet est de vérifier que lorsque le programme est lancé, les différentes configurations de jeu sont disponibles.

– **Test 6**

Essayer chaque action possible dans le menu lance et vérifier que l'on obtient la configuration attendue.

2.4.3 Types de joueurs

On vérifiera que des moyens de modularités permettant d'interchanger la façon dont l'ordinateur joue ont été mis en place (voir section intégration).

On implémentera alors si possible de nouveaux niveaux pour vérifier cette modularité.

– **Test 7**

Test exhaustif d'exécution vérifiant qu'au moins deux niveaux de jeu sont présents.

2.4.4 Visualisation

Les tests de visualisation n'étant pas l'objectif de ce projet nous résumerons les tests à effectuer comme suivant, et ne présenterons pas de résultats dans la partie correspondante. On admettra que l'interface graphique sera exempte de bug dans la suite du rapport.

On vérifiera que lorsque le programme est lancé, il est possible à tout moment de jouer l'ensemble des coups autorisés. Ceci peut être effectué à la fois manuellement et automatiquement grâce à un mode ordinateur aléatoire. Rappelons ici qu'un joueur ordinateur aléatoire ne devra alors être présent que dans une situation de test et non pas dans la version livrable du programme. Ce type de joueur devra être testé séparément pour s'assurer que les conclusions tirées de son utilisation ne sont pas faussées par nature.

Dans le cas d'une interface graphique, chaque bouton sera testé dans une démarche exhaustive.

Il est nécessaire de rappeler qu'en raison de la simplicité du jeu ici modélisé, les tests systèmes ne représentent pas une difficulté élevée et il sera trivial de vérifier la plupart de ces spécifications dans la mesure où les tests unitaires et les tests d'intégration (décrits plus tard) seront passés avec succès.

3 Architecture

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

3.1 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.

3.2 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.

3.2.1 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.

3.2.2 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 booléen. 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.

3.3 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 détection 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 détection 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 événements 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.

3.4 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 apparaisse.

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.

3.5 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 permettront 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.

3.6 Tests d'intégration

Une fois les tests JUnit terminés, il nous faudra prévoir des tests d'intégrations. Nous saurons alors que chaque partie fonctionnera correctement indépendamment.

=====

SCHEMA A DEPLACER

=====

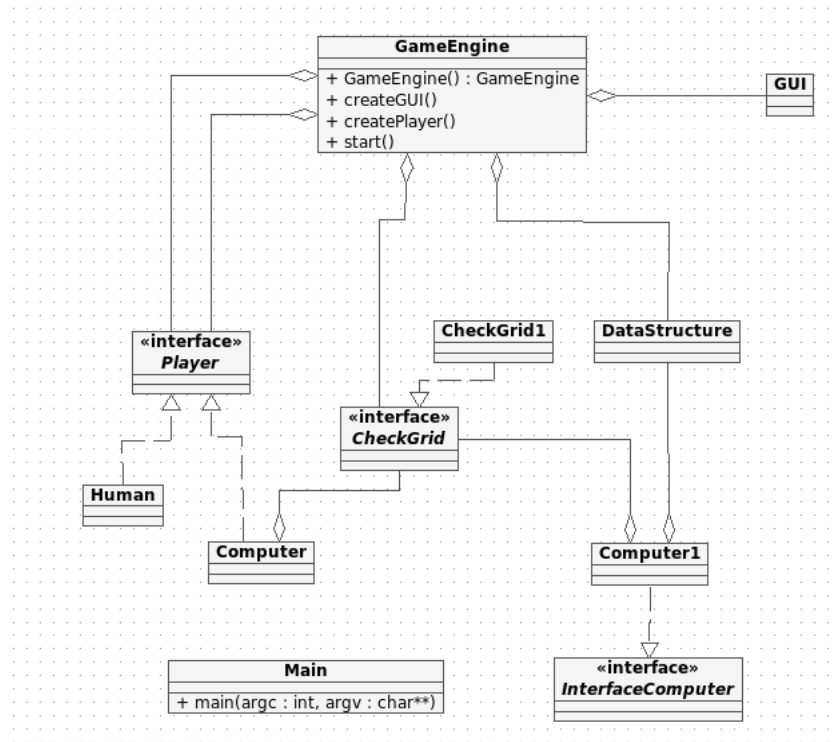


FIG. 1 – Notre première architecture

Moteur de jeu - interaction avec les autres modules.

Nous sommes conscient que ce sera notre moteur de jeu qui va prendre le plus de temps pour nos tests d'intégrations. C'est lui qui va instancier notre structure de données, nos joueurs, nos règles ainsi que l'interface graphique.

C'est pourquoi nous ne mettrons dans cette partie que les données de tests qui concernent les différents modules avec le moteur de jeu.

Structure de données - instanciation et modification(s)

- Instanciation correcte

Lors du lancement du jeu, notre moteur va instancier notre structure de données. Nous avons déjà pensé que ce sera une matrice, qui devra être, d'après les spécifications, de taille minimale 6×7 . Nous devons vérifier que lorsqu'une grille correcte est instanciée, l'objet retourné soit un objet de la taille souhaitée.

- Instanciation incorrecte

Dans un soucis de robustesse, nous pensons traiter le cas où une grille de taille incorrecte soit instanciée par notre moteur de jeu. Autrement dit une grille de taille inférieure à

6×7. Pour mieux comprendre, si un utilisateur vient à modifier notre code pour instancier une grille trop petite, notre structure de données doit en tenir compte et construire une grille de taille minimale 6×7.

- Lecture/Modification correcte

Lors d’une partie notre moteur de jeu doit pouvoir lire et modifier sans difficulté la grille. Nous devons tester si les modifications apportées par notre moteur de jeu soient bien effectuées par notre structure de données et comparer les valeurs que notre moteur de jeu lit dans notre structure de données avec celles renvoyées par notre structure de données.

- Lecture/Modification incorrecte

Encore une fois, dans un souci de robustesse, la structure de données devra pouvoir traiter des saisies incorrectes. Pourtant cela ne veut pas dire qu’elle devra les interpréter, mais plutôt qu’elle signifiera à l’utilisateur qu’il y a eu une mauvaise saisie à travers la console, et renverra un booléen `false`.

Joueurs - instanciation et interaction

- Instanciation

Lorsque nous créons un joueur, il ne peut être que créé correctement. Nous ne trouvons pas encore d’utilité à un constructeur prenant en compte des paramètres. La seule chose qui sera à vérifier, sera le type de l’instance du joueur. À savoir si c’est bien un ordinateur, ou alors un humain.

- Interaction

Nous devons vérifier que les coups joués par l’ordinateur sont bien ceux récupérés par le moteur du jeu. Il faudra aussi vérifier que si l’ordinateur renvoie une valeur impossible, par exemple une colonne pleine, cela ne perturbe pas le fonctionnement du programme, mais demande une nouvelle valeur à l’ordinateur. De la même manière on devra vérifier que la grille fournie à l’ordinateur par le moteur de jeu, corresponde à celle du moteur de jeu. Et une fois que l’ordinateur a joué, il faudra vérifier que la grille n’ait pas été modifiée par notre intelligence artificielle, car ce sera le job de notre moteur de jeu.

Règles - instanciation et utilisation

- Instanciation

Nos règles ne prennent aucun argument, donc leur instanciation ne devraient pas poser de problème.

- Utilisation correcte

Il faudra soumettre aux règles une grille correcte et vérifier qu’elles ne modifient pas cette grille. Dans le cas où il y a un gagnant nos règles devront nous le spécifier.

- Utilisation incorrecte

Nous espérons avoir un module robuste pour gérer les accès incorrects fait par le moteur de jeu, mais aussi des grilles incorrectes. Avec plusieurs gagnants, et encore une fois voir si nos règles valident ou non ces grilles.

Notons que notre moteur de jeu utilisera toutes les méthodes de notre structure de données, ainsi que de nos règles. Donc si l'interaction entre notre moteur de jeu, notre structure de données et nos règles sont correctes, alors si un autre module utilise de la même manière la structure de données et les règles, et bien ce dernier module fonctionnera sans modifications ni tests nécessaires.

Ce cas devrait surement s'appliquer à notre intelligence artificielle qui devrait être capable de calculer des positions possibles. Pour se faire, l'intelligence artificielle devra interagir avec les règles et aussi la structure de données.

Et enfin, pour vérifier de nouveau, que notre logiciel marche, en d'autres termes que nos modules interagissent correctement, nous pensons intégrer un mode débogue qui ne serait accessible qu'en mode de test.

4 Implementation

4.1 DataStructure

4.1.1 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 être compris entre 0 et respectivement **height** et **width**.

Nous n'avons pas mis de restriction sur **color**, étant 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 apportée 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.

4.1.2 Tests de la matrice

Nos différentes classes de test vont dériver de **junit.framework.TestCase**. Elles auront 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'auront pas de paramètres et retournent **void**. Leur nom commencera par **testXXX**.

A l'intérieur des méthodes de test, on effectuera les actions de test souhaitées, et on vérifiera

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

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

Le premier test envisagé sera 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 réalisera 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 limites sera 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));
```

4.2 Player

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

4.2.1 HumanPlayer

```
private int currently_played;
```

Sert à mémoriser la position que le joueur a choisi.

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

Cette méthode va attendre que le joueur fasse 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`.

4.2.2 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 attribue donc un `mode` au `CpuPlayer` mais aussi les règles à 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 choisi. 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 totalement indépendante du reste du programme et peut être changé facilement.

4.2.3 Jeux de test

Tester la classe `HumanPlayer` se limitera à tester la méthode `play(DataStructure , GUI gui)`.

Nous allons tester que la valeur de la position jouée n'est pas nulle et qu'elle est comprise entre 0 et la largeur de la grille(7), cela s'écrira :

```
played = human.play(matrix,app) ;  
assertNotNull(played) ;  
assertTrue( 0 <= played && played < matrix.getWidth());
```

4.3 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.

4.3.1 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 implémenter 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.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|--------|-----|--------|--------|---|---|
| | | | | | | |
| | Yellow | | | | | |
| | Red | | | | | |
| | Red | Red | Yellow | | | |
| | Yellow | Red | Red | Yellow | | |

FIG. 5 – playable[i] = 3

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

[illegible]

FIG. 6 – 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.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | | | | | | | |
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |

FIG. 7 – 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`.

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()` n'est 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 fait 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 d'en 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.

4.3.2 Jeux de test

Nous allons reprendre les exemples de la partie implémentation `IaForInRow` en ajoutant comme prévu de nouveaux cas.

Les données de tests se composent d'un ordinateur avec un mode (facile ou difficile) et de l'état de la structure de données, c'est à dire de la grille de jeu. Dans le rapport l'ensemble des tests se fera à l'aide d'une IA en mode difficile. L'IA facile ayant le même comportement avec des stratégies en moins (méthodes en moins), il n'est pas nécessaire de la tester. Certains tests sont quand même présents dans `Junit` sur l'ordinateur en mode facile avec des grilles

non conventionnelles. De même la validité des accesseurs a été testé. Nous avons choisis ici de présenter que les résultats significatifs.

L'ordinateur joue de manière séquentielle, si bien qu'il est possible de calculer chacun de ses coups en connaissant l'état de sa grille `playable[i]`.

Tout d'abord nous avons vérifié que `initialize(DataStructure grid, int difficulty)` initialise correctement l'objet `IaForInRow`. C'est à dire que `CpuGrid` soit le même Objet que la grille `grid` passée en référence, et que le mode de difficulté soit correct.

```
//Standard grid and easy difficulty
ia1.initialize(grid, mode1);
[...]
assertNotNull(ia1.getCpuGrid());
assertNotNull(ia1.getWidth());
assertNotNull(ia1.getHeight());
assertNotNull(ia1.getMode());
assertSame(ia1.getCpuGrid(),grid);
assertEquals(7,ia1.getWidth());
assertEquals(6,ia1.getHeight());
assertEquals(1,ia1.getMode());
```

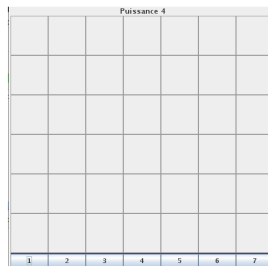


FIG. 10 – Grille vide

Avec une grille vide nous devons vérifier que l'ordinateur ne met pas en place de stratégie, donc que `playable[i]` est remplie de 0 et ainsi que, par défaut, l'ordinateur joue au centre de la grille.

```
int Ia2Played = ia2.play(rule);
assertNotNull(Ia2Played);
assertTrue(0 <= Ia2Played && Ia2Played < grid.getWidth());
assertEquals(3, Ia2Played);
int[] playable2 = ia2.getPlayable();
for ( int i =0 ; i<ia2.getCpuGrid().getWidth();i++)
assertEquals(0,playable2[i]);
```

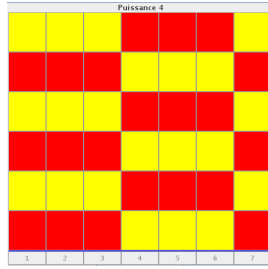


FIG. 11 – Grille pleine

La grille étant pleine l'ordinateur ne peut pas jouer. Elle retourne la valeur -1 qui dit qu'aucune position n'est jouable. Cette valeur n'est pas interprétée par le GameEngine, car il ne se produira jamais (voir analyse statique).

```
int Ia2PlayedFullGrid = ia2.play(rule);
assertFalse(0 <= Ia2PlayedFullGrid && Ia2PlayedFullGrid < grid.getWidth());
assertEquals(-1, Ia2PlayedFullGrid);
```

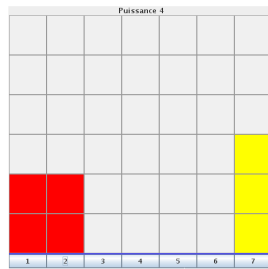


FIG. 12 – Grille avec coup gagnant

Si il a une possibilité de gagner, il repère la colonne qui permet de gagner en y mettant le code `playable` à 3 pour ensuite y jouer.

```
int Ia4PlayedForWin = ia4.play(rule);
assertNotNull(Ia4PlayedForWin);
assertTrue(0 <= Ia4PlayedForWin && Ia4PlayedForWin < grid_null.getWidth());
assertEquals(6, Ia4PlayedForWin);
int[] playable4 = ia4.getPlayable();
for ( int i =0 ; i<ia4.getCpuGrid().getWidth();i++)
    if ( i == 6)
assertEquals(3,playable4[i]);
    else
assertEquals(0,playable4[i]);
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|--------|--------|--------|--------|---|---|
| | | | | | | |
| | Yellow | | | | | |
| | Red | | | | | |
| | Red | Yellow | Yellow | | | |
| | Yellow | Red | Red | Yellow | | |

FIG. 13 – Autre cas de coup gagnant

L'ordinateur peut gagner en jouant en 2, et il doit le faire avec `playable[2]=3`.

```
int Ia2Playedfig4 = ia2.play(rule);
int[] playable7 = ia2.getPlayable();
assertEquals(2 , Ia2Playedfig4);
assertEquals(3, playable7[2]);
```

[illegible]

FIG. 14 – Colonne injouable

Si l'ordinateur joue sur la colonne 4, alors au prochain coups l'autre joueur pourra gagner avec une diagonale. L'ordinateur devra avoir `playable[4]=1` qui correspond a un placement de pions qui fait gagner l'adversaire. Dans ce cas nous savons que l'ordinateur va jouer en 2 du au fait du caractère non aléatoire de l'ordinateur et surtout que `playable[4]=1`.

```
int Ia2Playedfig2 = ia2.play(rule);
int[] playable5 = ia2.getPlayable();
assertEquals(2 , Ia2Playedfig2);
assertEquals(1 , playable5[4]);
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|--------|-----|--------|--------|---|---|
| | | | | | | |
| | Yellow | | | | | |
| | Red | | | | | |
| | Red | | Yellow | Red | | |
| | Yellow | Red | Red | Yellow | | |

FIG. 15 – Stratégie sur plusieurs coups

Si l'ordinateur joue en 2 cela permettra à l'autre joueur de bloquer un éventuel coup gagnant futur. L'ordinateur doit avoir `playable[2] = 2` ce qui est un code moins fort que 1 mais il ne doit pas y jouer tout de même dans la mesure du possible. Si il doit choisir entre un `playable` à 1 ou 2 il choisira tout de même le code 2.

```
int Ia2Playedfig3 = ia2.play(rule);
int[] playable6 = ia2.getPlayable();
assertEquals(3 , Ia2Playedfig3);
assertEquals(2, playable6[2]);
```

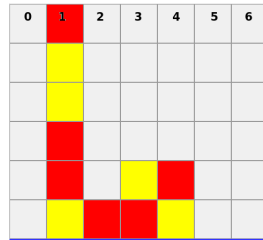


FIG. 16 – Colonne injouable

La colonne 1 est pleine donc l'ordinateur ne doit plus y jouer et `playable[1] = 4`. C'est le code qui interdit de jouer le plus fort. Si toutes les colonnes sont à 4 il renverra un code d'erreur -1. C'est le seul code de `playable` ou il est totalement impossible de jouer dans la colonne marquée.

```
int[] playable8 = ia2.getPlayable();
assertEquals(4, playable8[1]);
```

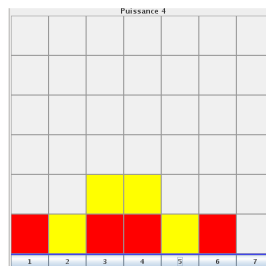


FIG. 17 – Strategie sur plusieurs coup

L'ordinateur doit remarquer qu'il peut gagner en deux coup il met donc `playable` à 5 sur les cases offrant cette possibilité : `playable[2] = 5` et `playable[5] = 5`. (colonne 2 et 6 sur la grille). Il prépare donc comme voulu sa stratégie sur 2 coups qu'il modifiera en fonction de du coup jouer par l'autre joueur.

```
int Ia2Playedfig6 = ia2.play(rule);
```

```
int[] playable9 = ia2.getPlayable();
assertEquals(5, Ia2Playedfig6);
assertEquals(5, playable9[1]);
assertEquals(5, playable9[5]);
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | |
| 6 | | | | | | | |
| 5 | | | | | | | |
| 4 | | | | | | | |
| 3 | | | | | | | |
| 2 | | | | | | | |
| 1 | | | | | | | |
| 0 | | | | | | | |

FIG. 18 – Stratégie de blocage

L'ordinateur peut aussi prévoir à l'avance les positions dangereuses des pions de l'autre joueur, il mettra ainsi en place la tactique **playable** 6 dans les colonnes qui couperont les alignements possibles des pions adverses. Dans ce cas la **playable**[1] = 6 et **playable**[4] = 6. Il jouera donc dans l'une de ces deux colonnes, ici la colonne 4.

```
int Ia2Playedfig7 = ia2.play(rule);
int[] playable10 = ia2.getPlayable();
assertEquals(4 , Ia2Playedfig7);
assertEquals(6, playable10[0]);
assertEquals(6, playable10[4]);
```

[illegible]

FIG. 19 – Coup Obligatoire

L'ordinateur est obligé de jouer en colonne 4 sous peine de voir l'autre joueur gagner au prochain coup. Nous devons donc avoir `playable[4]=6` qui est le code prioritaire pour les coups à jouer après 3.

```
int Ia2Playedfig8 = ia2.play(rule);
int[] playablefig8 = ia2.getPlayable();
assertEquals(4 , Ia2Playedfig8);
assertEquals(6, playablefig8[4]);
```

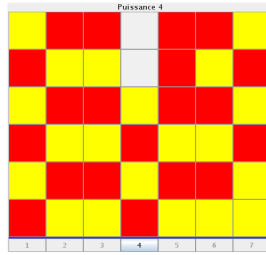


FIG. 20 – Non blocage du jeu

Il peut arriver que l'ordinateur n'est pas le choix et doit jouer dans une colonne même si son `playble` est à 1 c'est a dire que jouer ce coup fera gagner l'autre joueur, nous devons vérifier que l'ordinateur le fera quand même sous peine de bloquer la partie. Donc ici `played[3]=1` , mais l'ordinateur joue en 3 (4ème colonne) et fait ainsi gagner l'autre joueur.

4.4 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 ...

4.4.1 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 c'est jouable, et `false` sinon.

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

4.4.2 Tests de FourInARow

Cette classe testera :

- L’existence ou non d’un alignement de 4 jetons d’une même couleur soit dans la même ligne, même colonne, même diagonale :
On choisira 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 créera 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 arrivera 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 soit jouable/non jouable.

4.5 GameEngine

4.5.1 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 être 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 fait 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 boutons 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 pas 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 c'est elle qui va générer la gravité, si on peut dire.

4.5.2 Tests du GameEngine

Nous devons élaborer des tests fonctionnels pour la classe `GameEngine` afin de vérifier la conformité du déroulement du jeu et les données retournées par la classe.

Les méthodes seront `testHuman2WinHuman1()`, `testCpu1WinHuman()` et `testCpu2WinHuman()` qui lancent respectivement des confrontations Humain/Humain, Cpu1/Humain et Cpu2/Humain. Elles auront pour rôle respectivement de :

- Tester que lors de la confrontation Humain1/Humain2 c'est le joueur humain2 qui a gagné et envoyer un message d'erreur dans le cas contraire.

```
assertFalse("2 possibilités : \n C'est le joueur humain numéro 1 qui a  
gagné !!!\n" + "ou personne n'a gagné !!", game.getWinPlayer());
```

- Tester que lors de la confrontation Cpu1/Humain c'est le joueur Cpu1 qui a gagné et envoyer un message d'erreur dans le cas contraire.

```
assertFalse("C'est l'humain qui a gagné!", game.getWinPlayer());
```

- Tester que lors de la confrontation Cpu2/Humain c'est le joueur humain2 qui a gagné et envoyer un message d'erreur dans le cas contraire.

```
assertFalse("C'est l'humain qui a gagné!", game.getWinPlayer());
```

Une dernière méthode `testModes()` va tester la mise-à-jour des modes de jeu.

Exemple :

```
assertTrue(game.initMode(0));
assertTrue(game.getMode() == 0);
assertFalse(game.getMode() == 1);
assertFalse(game.getMode() == 2);
```

4.6 Main et Menu

Le `Main()` va instancier le `GameEngine` mais aussi le menu de départ. c'est 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.

4.7 GUI

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

4.7.1 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 affiche le nom du joueur dans l'interface graphique, et la seconde affiche Match nul.

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

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

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

`greyButton()` grise le bouton 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.

5 Analyse Statique

6 Résultats

6.1 Tests de validation

Nous allons voir ici les résultats des tests imaginés pour s'assurer que le programme obtenu respecte bien les termes précisés dans la spécification.

6.1.1 Règles de puissance 4

– **Test 1**

Lancer Humain vs Humain : succès.

Lancer Humain vs Cpu niveau 1 : succès.

Lancer Humain vs Cpu niveau 2 : succès.

– **Test 2**

Pour éprouver le programme nous avons défini une relation d'équivalence comme tel : Deux données de tests $dt1$ et $dt2$ (ici des entiers représentant les 2 dimensions de la grille) sont en relation : $dt1 \diamond dt2$ si et seulement si leur valeur appartient au même intervalle $I \in N$. On forme une partition de ces intervalles sur l'ensemble des valeurs logiques d'instantiation. On entend par valeur logique d'instantiation une valeur susceptible d'être instanciée car elle permet de jouer une partie (i.e on n'instanciera jamais une grille de dimensions inférieures à 4). Voici les intervalles concernés :

On suppose que dtH représente la hauteur de la grille, dtL la largeur. Les valeurs par défaut étant $dtH = 6$ et $dtL = 7$, on va s'intéresser aux intervalles inférieurs et supérieurs et lancer les tests sur chaque combinaison des classes d'équivalences formées par la méthode expliquée ci-dessus. Ainsi, pour la hauteur, on obtient les intervalles $I_{H1} = [4, 6]$, $I_{H2} =]6, +\infty]$ et pour la largeur, $I_{L1} = [4, 7]$, $I_{L2} =]7, +\infty]$.

On choisit alors les combinaisons suivantes : $dtH = 4$, $dtL = 5$, $dtH = 15$, $dtL = 6$, $dtH = 5$, $dtL = 20$, $dtH = 9$, $dtL = 9$.

Chacune de ces instantiations a été réalisée avec succès et le comportement du programme correspondait à nos attentes.

– **Test 3**

On utilise le mode humain vs humain pour arriver à une partie nulle et on vérifie que l'on est averti de cette issue.

On soumettra ensuite le programme à un échantillon d'utilisateurs connaissant les règles et on recueillera leur avis concernant leur application.

– **Test 4**

Demander aux personnes composant l'échantillon si elles jugent que les règles de puissance 4 attendues ont été respectées pendant leur partie.

Enfin, on pourra utiliser le joueur ordinateur pour générer des parties aléatoires et en vérifiant à chaque fois que l'empilement des pions respecte les règles.

- **Test 5**
Générer des parties aléatoires et vérifier les coups joués.

6.1.2 Configurations de jeu

Il suffit d'un test fonctionnel basique dont l'objet est de vérifier que lorsque le programme est lancé, les différentes configurations de jeu sont disponibles.

- **Test 6**
Essayer chaque action possible dans le menu lance et vérifier que l'on obtient la configuration attendue.

6.1.3 Types de joueurs

On vérifiera que des moyens de modularités permettant d'interchanger la façon dont l'ordinateur joue on été mis en place (voir section intégration).

On implémentera alors si possible de nouveaux niveaux pour vérifier cette modularité.

- **Test 7**
Test exhaustif d'exécution vérifiant qu'au moins deux niveaux de jeu sont présents.

6.2 Tests d'intégration

Nous n'avons pas codé l'interface graphique directement, elle n'est venue que dans un second temps. Autrement dit pour jouer au puissance 4 on passait par la console.

Mais avant d'en arriver là nous avons intégré notre **GameEngine** avec notre **DataStructure**.

La simplicité, à ce moment, de ces deux méthodes étaient telles, que les tests d'intégrations ne nous ont pas vraiment permis, à ce moment là, d'améliorer le code.

Dans un second temps, nous avons créé un module **Rule** que nous avons intégré à notre **GameEngine**. Nous avons rencontré de nombreux problèmes. Car notre **Rule** a besoin de la grille pour pouvoir vérifier si la grille est valide et surtout si il y a un gagnant. Pour cela nous avons dû créer des accesseurs pour les variables **private** de **DataStructure**.

L'étape d'après nous avons mis en place un joueur qui jouait à travers la console. Encore une fois dans un soucis de tests d'intégrations pour vérifier l'affichage (primaire) de la grille dans notre console. Cette étape nous a permis de voir plusieurs choses.

- Dérouler le jeu en séquence

Il fallait savoir qui jouait et donc alterner les deux joueurs.

- Déterminer la fin du jeu

Il fallait faire des appels ingénieux à **Rule** pour déterminer si il y a un vainqueur, mais aussi si la partie est terminée, match nul. Pour ce dernier cas, nous avons choisi la solution de facilité. Nous avons mis en place un compteur dans le **GameEngine**, qui compte le nombre de coup joué. Et si ce compteur atteint la dimension de la matrice, alors ça veut dire que chaque case de la grille a été modifiée au moins une fois.

- Vérifier la validité du coup joué

Ca n'a pas vraiment été ça qui nous a posé problème, mais plutôt le fait que si nous jouions mal il fallait refaire jouer le même joueur et ne pas incrémenter le compteur. La seconde partie de la dernière phrase ne nous a pas paru évidente, c'est pourquoi nous avons dû tester plusieurs cas.

- Les deux joueurs jouent toujours correctement et un des gagnent.
- Les deux joueurs jouent toujours correctement mais ça fini par un match nul
- Les deux joueurs ne jouent pas correctement et il y avait match nul

Ce dernier test nous a permis de voir que nous avons fait une bourde au niveau du compteur, qui fut corrigé sur l'instant.

A ce niveau d'intégration là, le programme fonctionnait correctement. On pouvait jouer au puissance 4 a deux joueurs sans problème. L'étape suivante a consisté à mettre en place une interface graphique. D'après le sujet cette interface graphique doit être interchangeable en ne modifiant qu'une seule ligne de code dans notre **GameEngine**. Pour respecter cette règle, nous avons mis une interface dont notre interface graphique hérite. Ça nous a fait remanipuler notre architecture, mais cela correspondait déjà en partie mieux au sujet.

Notre interface graphique est assez rudimentaire, les tests qui ont été effectués ont été basique. Nous avons effectué les mêmes tests que lors de l'intégration de l'affichage console. Mais nous avons déduit de grosses améliorations sur le **GameEngine**. C'est lors de ces tests que nous avons ajouté les méthodes qui servent à griser les boutons qui correspondent à des colonnes pleines. Ainsi que les fonctionnalités qui consistaient à ajouter le nom du vainqueur sur

notre grille, mais également la possibilité de faire un reset (et cette fois on a pensé à remettre à zéro le compteur). Nous avons quand même détecté des bug, dont la source nous est encore inconnue mais que nous avons corrigé en ajoutant un `System.out.println()` ;. Car notre affichage ne fonctionnait pas sans ça. Il est possible d'améliorer cette partie en mettant un simple `System.out.flush()`, mais ça reste à tester.

En intégrant notre intelligence artificielle nous avons du modifier le `GameEngine` qui avait une méthode `start()` pour chaque `mode` (0, 1 et 2). L'intelligence artificielle s'est faite assez aisément, car elle se sert de `Rule` et d'une `DataStructure`. Lors de l'intégration de notre intelligence artificielle, le `GameEngine` était suffisamment robuste pour répondre à des cas d'erreurs, précédemment testé lors de l'intégration du mode console. Cependant, pour ajouter une intelligence artificielle il fallait modifier les classes `GameEngine` et `IAFourInARow` ce qui ne correspondait aucunement au sujet. Sur ce point nous avons du faire des efforts d'architecture.

En créant une interface `Player` nous pouvions instancier un joueur humain ou un ordinateur. Si c'était un ordinateur on instancierait un `IAFourInARow` et ça ne répondait pas à nos besoins lors de l'intégration.

Nous avons du remodifier le code et créer une nouvelle interface `Cpu` dont le but était de permettre l'intégration d'une nouvelle intelligence artificielle en ne modifiant qu'une seule ligne de code. Ça a apporté une modification simple et évidente sur notre `CpuPlayer`.

Avant on faisait :

```
IAFourInARow() cpu1 = new IAFourInARow() ;
```

Maintenant on fait :

```
Cpu() cpu1 = new IAFourInARow() ;
```

Le gros changement fut de faire implémenter `IAFourInARow` à notre interface `Cpu`.

Une fois cette modification apportée, on avait un code modulaire pour l'interface graphique et pour l'ordinateur. On a donc fait un peu de zèle en faisant de même pour `Rule` qui est devenue `FourInARowRule` et qui implémente l'interface `Rule`.

De cette manière on peut jouer à plusieurs jeux en changeant de règle (Puissance 5 par exemple).

6.3 Tests Unitaires

6.3.1 DataStructureTest

La plupart des bugs identifiés seront liés aux valeurs des positions de la grille. Particulièrement, une affectation d'une couleur à une position **hors limite** est source de bugs. Le fait d'affecter une couleur quelconque à la position **(0,9)** d'une grille (8,9) est typiquement le type de bugs qu'on aura à traiter.

Test 1 - Résultats

```
public void testInvalidSetValues() {
    assertFalse(matrix.setValue(10, 1, 0));
    assertFalse(matrix.setValue(1, 10, 1));
    assertFalse(matrix.setValue(10, 10, 2));
}
```

Ce test montre bien qu'il n'est pas possible d'affecter des couleurs **hors des limites** à la grille **matrix** (6,7) considérée.

Test 2 - Résultats

```
public void testNegativeMatrix() {
    int resultNH = negative_matrix.getHeight();
    assertEquals("la hauteur d'une negative_matrix n'est pas 6", resultNH, 6);

    int resultNW = negative_matrix.getWidth();
    assertEquals("la largeur d'une negative_matrix n'est pas 6", resultNW, 7);
}
```

Ce test montre que les paramètres (height , width) d'une grille "négative" n'ont de valeur que respectivement 6(hauteur) et 7(largeur).

Ce même type de test sera expérimenté sur une grille "zero" et donnera le même résultat.

Test 3 - Résultats

Tester **aux limites** la possibilité d'affecter des couleurs à une grille (6,7).
Ce test peut être généralisé à des grilles (i,j) avec $i > 0$ et $j > 0$.

En effet, l'affectation des couleurs dans le Domaine des positions [0...42] réussit avec succès. Mais, pour les valeurs -1 et 43 le test montre une impossibilité d'affectation.

6.3.2 IAfourInARow

Ces tests nous ont permis de révéler de nombreux bugs liée aux comportements des IA. Tout d'abord sur les stratégies de **playable[]** qui n'était pas toujours juste par rapport à l'état actuel de la grille,et aussi le façon de jouer des IA ne correspondait pas toujours à ce qui était attendu dut au fait des priorités des **playable**.

Pour chaque état de la grille ils nous étaient possible a l'aide de l'algorithme de connaitre l'état de `playable[]` et le coup que doit jouer l'IA ainsi nous avons corrigé les moteurs d'IA pour qu'ils nous donnent exactement les résultats escomptés dans les cas typiques ci-dessus.

L'ensemble des tests ci-dessus permettent de couvrir intégralement les arrêtes de l'Automate de flow de contrôle de l'IA en mode difficile (et donc de l'IA facile qui est une sous partie), nous sommes donc dans le cas d'une couverture TER2, la complexité du CFA ne nous permet pas d'obtenir des TER supérieur sans une démultiplication des tests.

7 conclusion