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 : M Rollet

Table des matières

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 x m (par défaut 6 x 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 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 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 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.

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 précedente, et ne génèrera 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 receuillera 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 modularité 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.

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 ou les tests unitaires et les tests d'intégration (décrits plus tard) seront passés avec succès.

3 Architecture

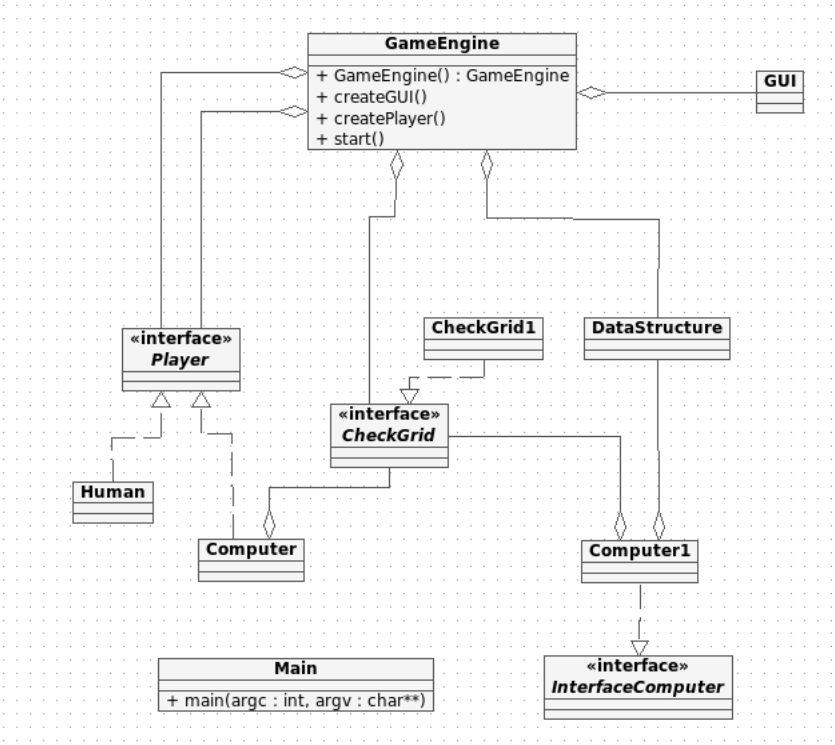

FIG. 1 – `Notre première architecture`

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.

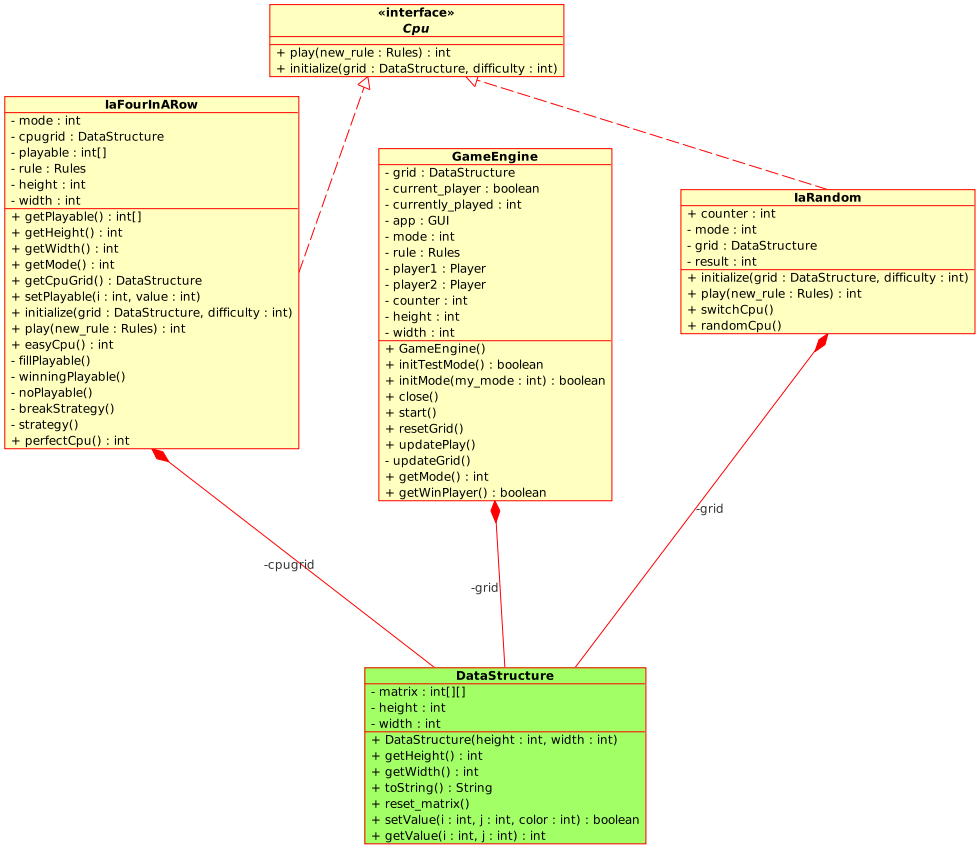

FIG. 2 – `Illustration des connexions de la structure de données`

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 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.

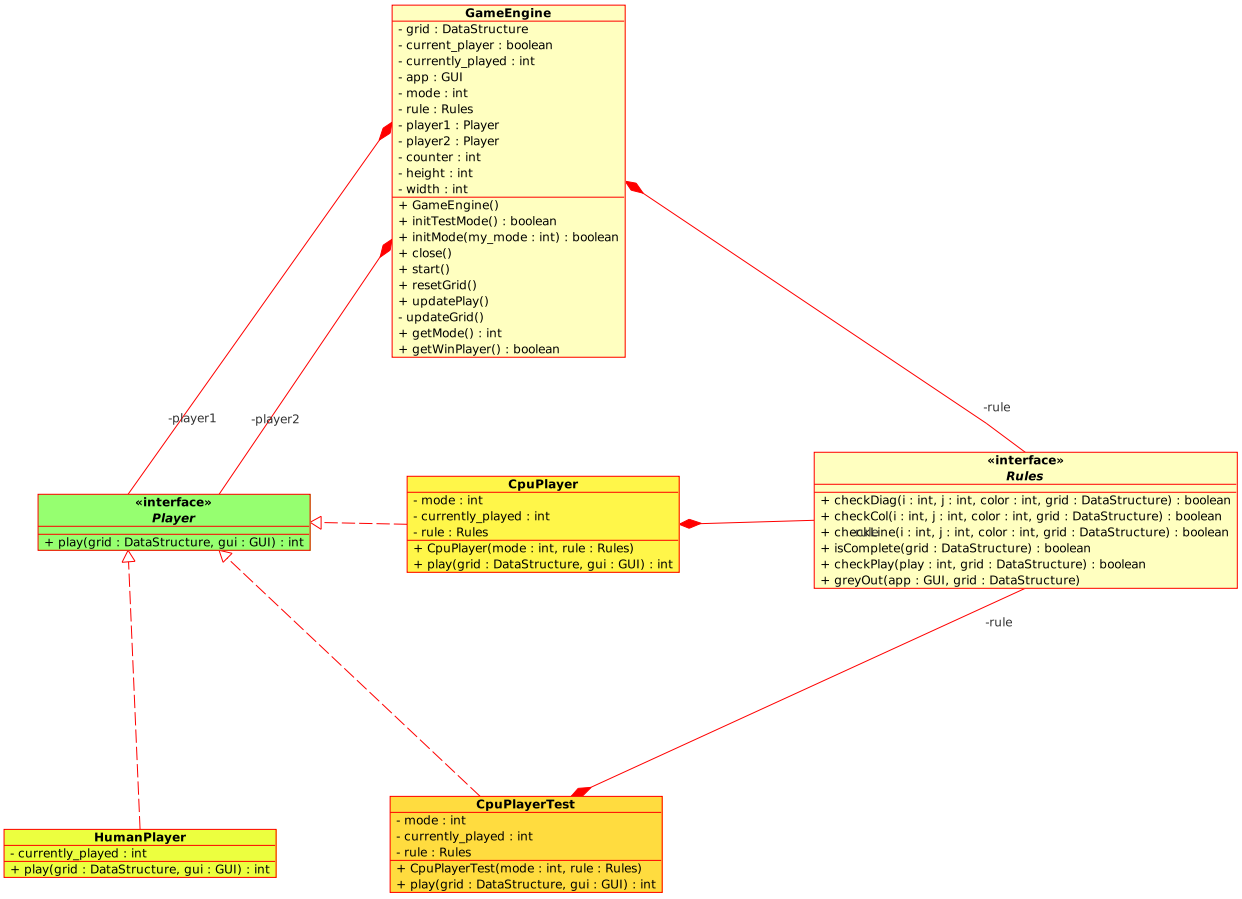

FIG. 3 – `Interactions des joueurs`

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 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.

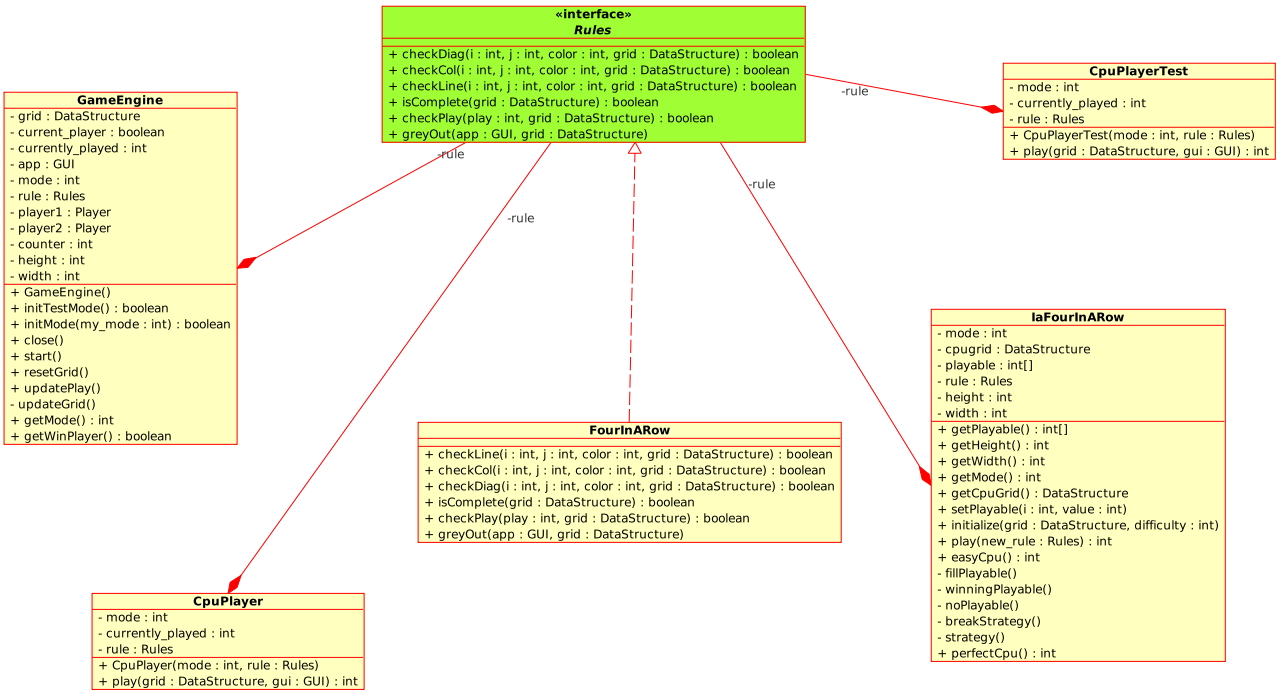

FIG. 4 – `Illustration des interactions du module de règles de jeu`

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 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.

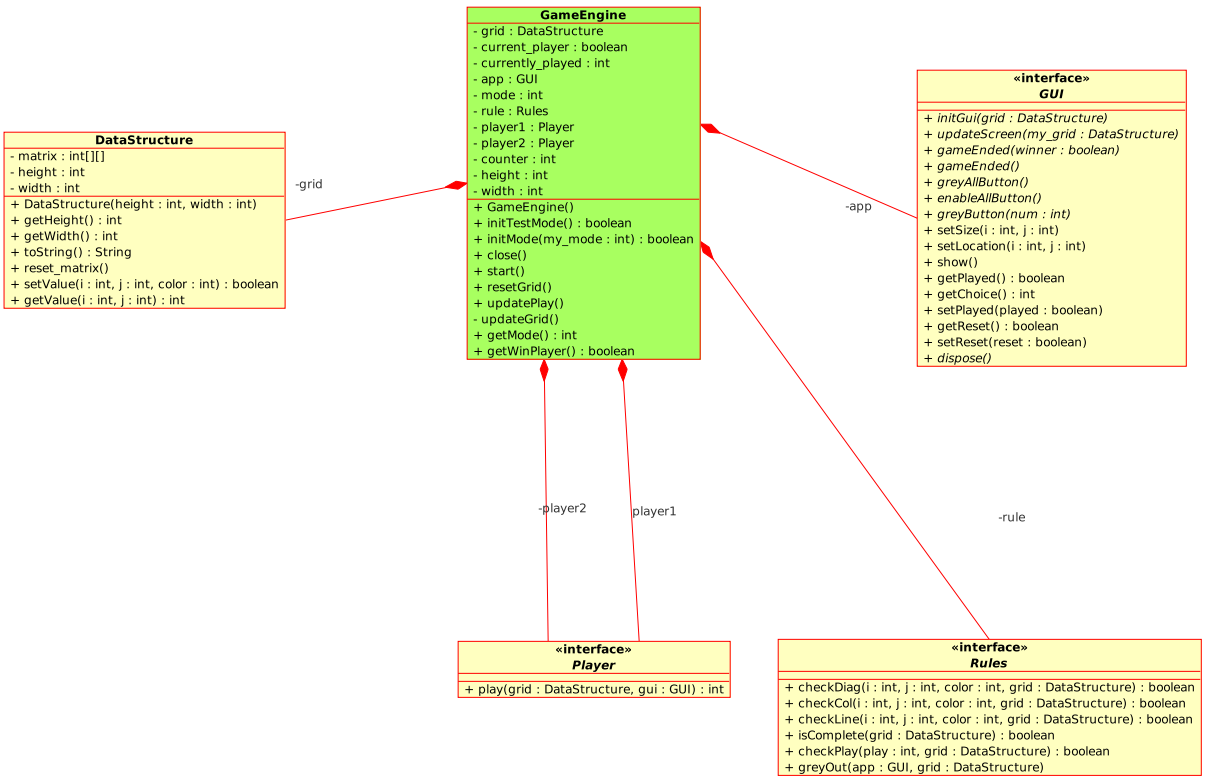

FIG. 5 – `Connexions du moteur de jeu`

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 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.

3.6 Tests d'intégration

Moteur de jeu - interaction avec les différents modules.

La classe GameEngine qui représente le moteur de jeu est centrale dans notre application. C'est le point de connexion de la plupart de nos modules. C'est cette classe qui va instancier la structure de données, les joueurs, les règles du jeu ainsi que l'interface graphique. On

concentre donc nos efforts de test d'intégration sur ces points de connexion. Les parties qui suivent porteront donc sur des tests entre chaque module et le moteur du jeu.

Structure de données - instanciation et modification(s)

- `Test d'instanciation correcte`
  Lors du lancement du jeu, le moteur va instancier la structure de données. On teste donc l'instanciation en rentrant des valeurs particulières pour la grille. On vérifie que l'objet créé a la taille demandée.

- `Test d'instanciation incorrecte`
  Dans un soucis de robustesse, on teste le cas où une grille de taille incorrecte est instanciée par le moteur de jeu.
- `Test de lecture/modification correcte`
  Lors d'une partie le moteur de jeu doit pouvoir lire et modifier sans difficulté la grille. Nous devrons tester si les modifications apportées par le moteur de jeu sont bien effectuées par la structure de données.

- `Test de lecture/modification incorrecte`
  Toujours dans un soucis de robustesse, la structure de données doit pouvoir traiter les saisies incorrectes. `false`.

Joueurs - instanciation et interaction

- `Test d'instanciation`
  Il n'y a pas d'erreur possible sur les paramètres du constructeur de joueur car ce dernier n'admet pas de paramètres. Le seul point qui doit être vérifié est le type de l'instance du joueur.

- `Tests d'interaction`
  Nous devons vérifier que les coups joués par l'ordinateur sont bien ceux récupérés par le moteur du jeu. Il faut aussi vérifier que si l'ordinateur renvoit une valeur impossible, par exemple une colonne pleine, cela ne perturbe pas le fonctionnement du programme, mais demande une nouvelle valeur à l'ordinateur.
- `Test d'intégrité de la grille`
  On doit vérifier que la grille fournit à l'ordinateur, corresponde a celle du moteur de jeu. On doit donc vérifier que le passage en paramètre de la grille au joueur ne modifie pas cette dernière.

Règles - instanciation et utilisation

- Séquence de jeu
  On doit tester si l'alternance des coups entre les deux joueurs est correcte.

- Intégrité de la grille
  Il faudra soumettre aux règles des grilles correcte et vérifier que la grille n'est pas modifiée. Dans le cas ou il y a un gagnant nos règles devront nous le spécifier.

- Condition de terminaison
  Les contitions de victoires ou match nul doivent être correctes. Ceci sera vérifié sur un nombre maximal de parties (manuellement ou automatiquement).

- 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 valides ou non ces grilles.

Modularité Il doit être possible d'implémenter une nouvelle interface ou un nouveau niveau de jeu en changeant une seule ligne de code. Si un tel changement nécessite une modification plus vaste le test n'est pas considéré comme réussi.

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` et respectivement `width`.

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

Le constructeur va prendre deux entiers en paramètres. Il va vérifier si les entiers sont valides c'est à qu'ils ne soient 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. C'est pourquoi nous avons implémenté des accesseurs. Ce ne sont pas des accesseurs directs. Il n'y a pas de risque de modification inattendues. La méthode `setValue` va modifier notre matrice en respectant la contrainte exigeant que `i` et `j` doivent être 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 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` affiche 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'écrira :

```
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.Elle retourne la position choisit par le joueur mais ne vérifie pas si elle est correcte. Cela s'effectue plus loin dans le `GameEngine`.

4.2.2 CpuPlayer

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

La première variable stocke 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`, stock les règles du jeu. Cela est utile pour l'ordinateur, afin de bien respecter les règles 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 un `mode` au CpuPlayer mais aussi les règles à utiliser (`Rules`).

Notre seconde méthode, `play`, est similaire à celle de HumanPlayer. Elle va 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 totallement indépendante du reste du programme et peut être changée 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'en avoir créé une va nous permettre de pouvoir implémenter de nouvelles IA sans modifier plus d'une ligne du code.

4.3.1 IaFourInARow

En ce qui concerne l'intelligence artificielle, nous avons choisi d'en implémenter une séquentielle. Il n'y a aucune part aléatoire. Cette IA n'implémente pas la stratégie gagnante. Nous nous sommes renseigné sur celle-ci et il faudrait implémenté un arbre faisant intervenir des statistiques de meilleurs coups à 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 satisfaisants.

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 comme 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 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 à déterminer quel colonne jouer ou quelle colonne ne pas jouer. Tout cela se fait à l'aide d'un codage simple.

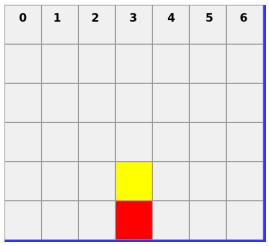

FIG. 6 – `playable[i] = 0`

Dans ce cas, tous les playable[i] sont à 0. Nous pouvons 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 cas.

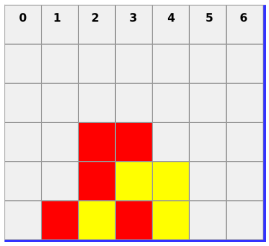

FIG. 7 – `playable[i] = 1`

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.

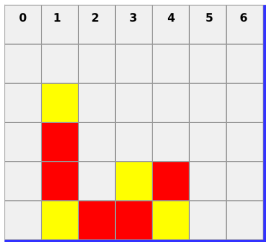

FIG. 8 – `playable[i] = 2`

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

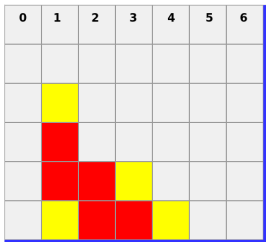

FIG. 9 – `playable[i] = 3`

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

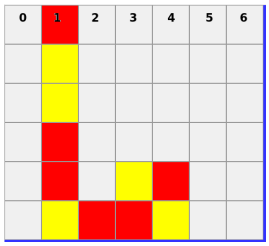

FIG. 10 – `playable[i] = 4`

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

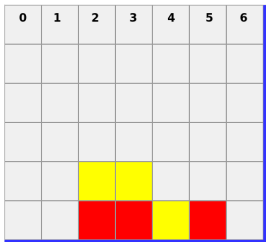

FIG. 11 – `playable[i] = 5`

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`.

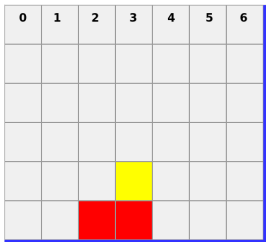

FIG. 12 – `playable[i] = 6`

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.

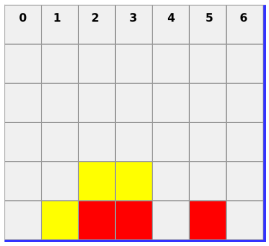

FIG. 13 – `playable[i] = 6`

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 joueur humain.

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

La première variable va contenir les règles du jeu 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 `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 dans 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, `playable[i] = 3`.

La méthode `fillPlayable()` quand a elle initialise `playable[i] = 0` pour les colonnes ou on peut encore jouer. En 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`, quel 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

Nos données de tests devront permettre de couvrir l'ensemble des retours de l'IA (mode facile et difficile) et des stratégies , c'est à dire :

- L'ensemble des coups possibles de l'ordinateur (toutes les colonnes de la grille).
- L'ensemble des valeurs de `playable` possible.
- Un retour d'erreur en cas d'impossibilitée de jouer.

Une donnée de test est composé de :

- Un grille vide , pleine , ou partiellement remplie.
- un mode de difficulté : deux sont prévus.
- Des règles de jeux (ici règles standard).

Pour cela nous allons reprendre les grilles d'exemples ci dessus afin de d'atteindre tout les valeurs possible de `playable`,en y ajoutant une grille pleine pour le code d'erreur. Nous devons tester ces grilles avec les deux modes de difficulté et avec les règles standard du puissance 4. Nous obtiendrons ainsi un jeu de test couvrant un maximum de cas d'état de l'IA et soulevant un maximum de comportement inattendut.

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 cést jouable, et `false` sinon.

La dernière méthode, `greyOut()`, va quand à elle griser les bouttons 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.

```
for (int i = -1; i <= matrix.getWidth(); ++i) {
for (int j = -1; j <= matrix.getHeight(); ++j) {
if (i == 0 && j>=0 && j < matrix.getWidth()- 2) {
assertTrue(rule.checkLine(i, j, 1, matrix));
assertFalse(rule.checkLine(i, j, 2, matrix));
} else {
assertFalse(rule.checkLine(i, j, 1, matrix));
assertFalse(rule.checkLine(i, j, 2, matrix));
}
}
}
```

- Tester qu'une position donnée est jouable en remplissant la grille aléatoirement mais de manière à ce qu'une position soit jouable/non jouable.

- Tester qu'un coup sur la ligne 0 de la grille n'est pas jouable s'écrira :

```
assertTrue(matrix.setValue(0, 0, 1));
assertTrue(matrix.setValue(0, 1, 1));
assertTrue(matrix.setValue(0, 2, 1));
assertTrue(matrix.setValue(0, 3, 1));
assertTrue(matrix.setValue(0, 4, 1));
assertTrue(matrix.setValue(0, 5, 1));
assertTrue(matrix.setValue(0, 6, 0));

assertFalse(rule.checkPlay(0, matrix));
```

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 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.

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. 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)`.

En 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 à `true` lorsqu'un joueur a choisi une colonne et est à `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 d'afficher/fermer une fenêtre. La méthode `dispose()` est aussi définit dans les 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

Dans cette partie, nous avons représenté des fonctions clés de notre programme à l'aide de CFA. Ceux-ci nous servent à effectuer une série de tests statiques des fonctions critiques du puissance4. Nous trouverons dans l'ordre : la fonction principale de notre moteur de jeu, puis dans une seconde partie les CFA représentant la partie IA du programme.

Pour plus de clarté les deux CFA principaux que sont `start` et `IA` sont amputés des fonctions auquel ils font appel. On considèrera dès lors que ces appels effectuent la fonction pour laquelle ils sont écrits et ne provoquent pas de bug dans le programme. Ces fonctions sont représentées par la suite et sont analysées séparément. Dans cette partie tous les tests sont effectués en boîte blanche.

5.1 GameEngine

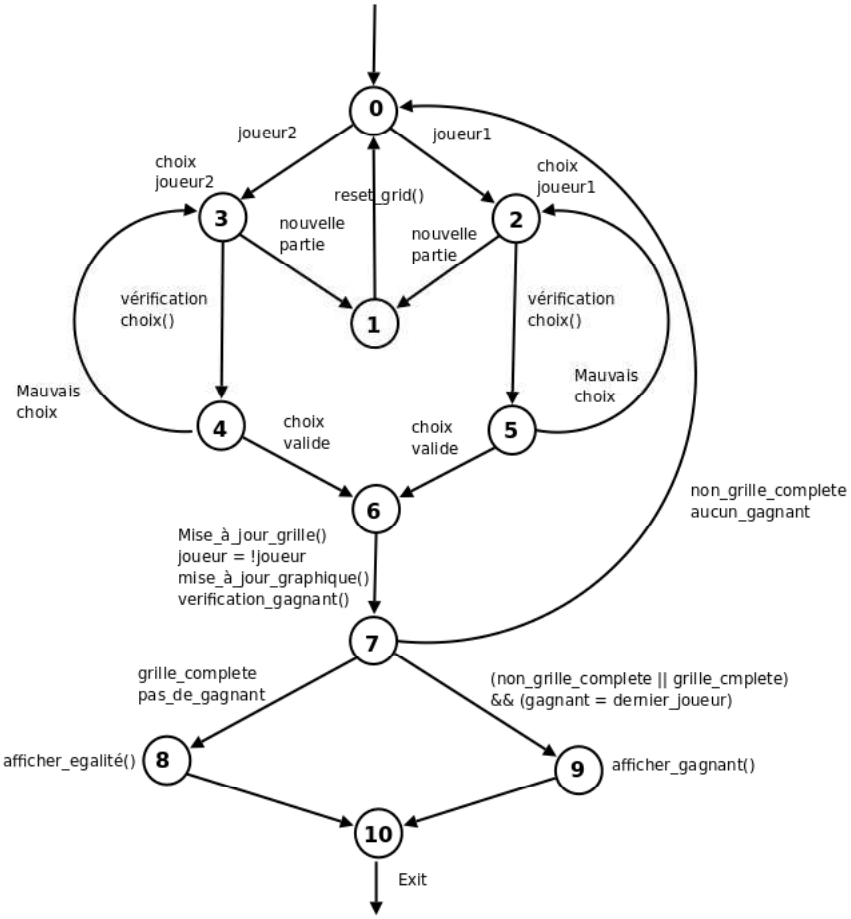

FIG. 14 – Fonction start

DT1={joueur1,joueur2,0=<choixJoueur1=<6,0=<choixJoueur2=<6,grille_vide} En supposant qu'aucun des deux joueurs ne réinitialise la partie, et que celle-ci se termine, nous obtenons deux chemins possibles :
{0-2-5-6-7-0-3-4-6-7-...-9-10} // partie avec un gagnant
{0-2-5-6-7-0-3-4-6-7-...-8-10} // partie avec un match nul

En prenant comme critère de couverture l'ensemble de tous les noeuds, nous obtenons un TER 1 = 9/11 = 81%.

DT2={joueur1,joueur2,choixJoueur1=aléatoire,choixJoueur2=aléatoire,grille_vide}

Si un des deux joueurs à un moment donné réinitialise la partie nous obtenons :
{0-2-5-6-7-0-3-4-6-7-...-0-2-1-0-...-9-10}
{0-2-5-6-7-0-3-4-6-7-...-0-2-1-0-...-8-10}

En prenant le même critère de couverture on a TER1=10/11= 90% ;

5.1.1 Les fonctions

`vérification_choix()`

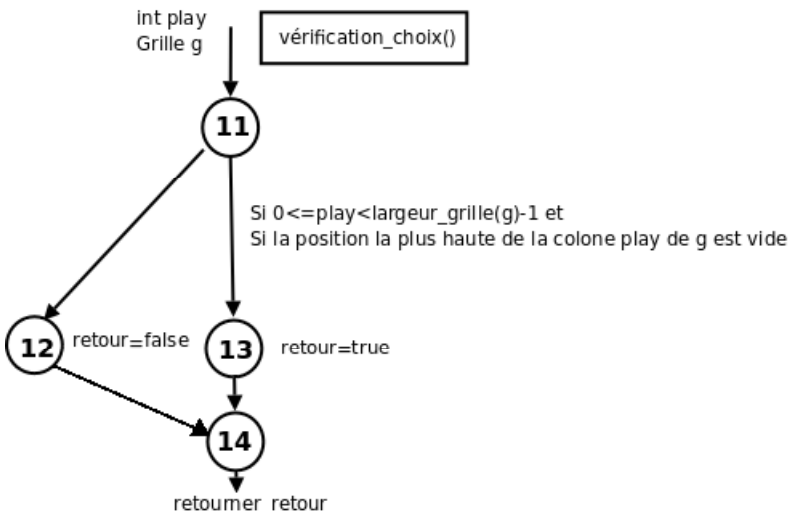

FIG. 15 – Fonction vérification choix

Cette fonction doit retourner un booléen servant à savoir si la valeur jouée est comprise entre les valeurs de la grille et si la colone dans laquelle on veut jouer est pleine.

DT1={0=<play=<6,grille_vide}
DT2={0=<play=<6,grille_pleine}
DT3={play=9,grille_vide}

Si nous prenons toujours le critère de couverture de l'ensemble de tous les noeuds, nous obtenons :

DT1 : TER1=3/4 = 75% {11-13-14}
DT2 : TER1=3/4 = 75% {11-12-14}
DT3 : TER1=3/4 = 75% {11-12-14}

`vérification_gagnant()`

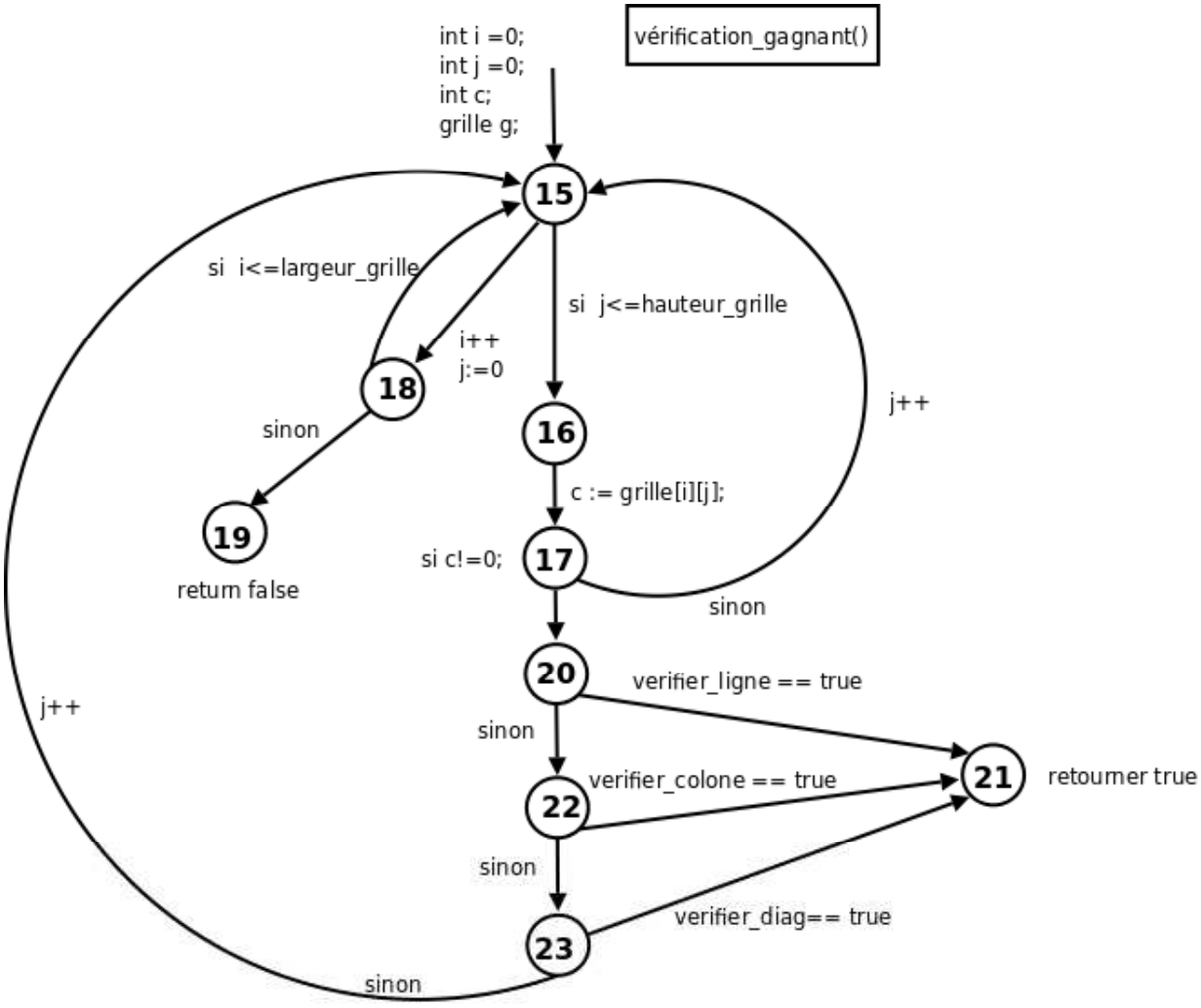

FIG. 16 – Fonction vérification gagnant

Pour ce CFA si dans les DT il existe une ligne gagnante, une colonne gagnante ou une diagonale gagnante, on admettra que l'assertion dans le programme retourne true.

DT1={grille_en_cours_ de_ partie,ligne_gagnante}
DT2={grille_en_cours_ de_ partie,colone_gagnante}
DT3={grille_en_cours_ de_ partie,diagonale_gagnante}
DT4={grille_en_cours_ de_ partie,pas_de_gagnant}

DT1={15-16-16-15-18-15-16-17-...-20-21}
DT2={15-16-16-15-18-15-16-17-...-20-22-21}
DT3={15-16-16-15-18-15-16-17-...-20-22-23-21}
DT4={15-16-16-15-18-15-16-17-...-15-18-19}

Critère de couverture : l'ensemble des sommets du graphe.

DT1 : TER1=6/9=66%
DT2 : TER1=7/9=77%
DT3 : TER1=8/9=88%
DT4 : TER1=5/9=55%

5.2 IA

Dans cette partie, nous avons seulement modélisé le CFA de l'IA en mode difficile car celui de l'IA facile est le même à la différence près qu'il ne comporte les fonctions *startegy*() et *breakStrategy*().

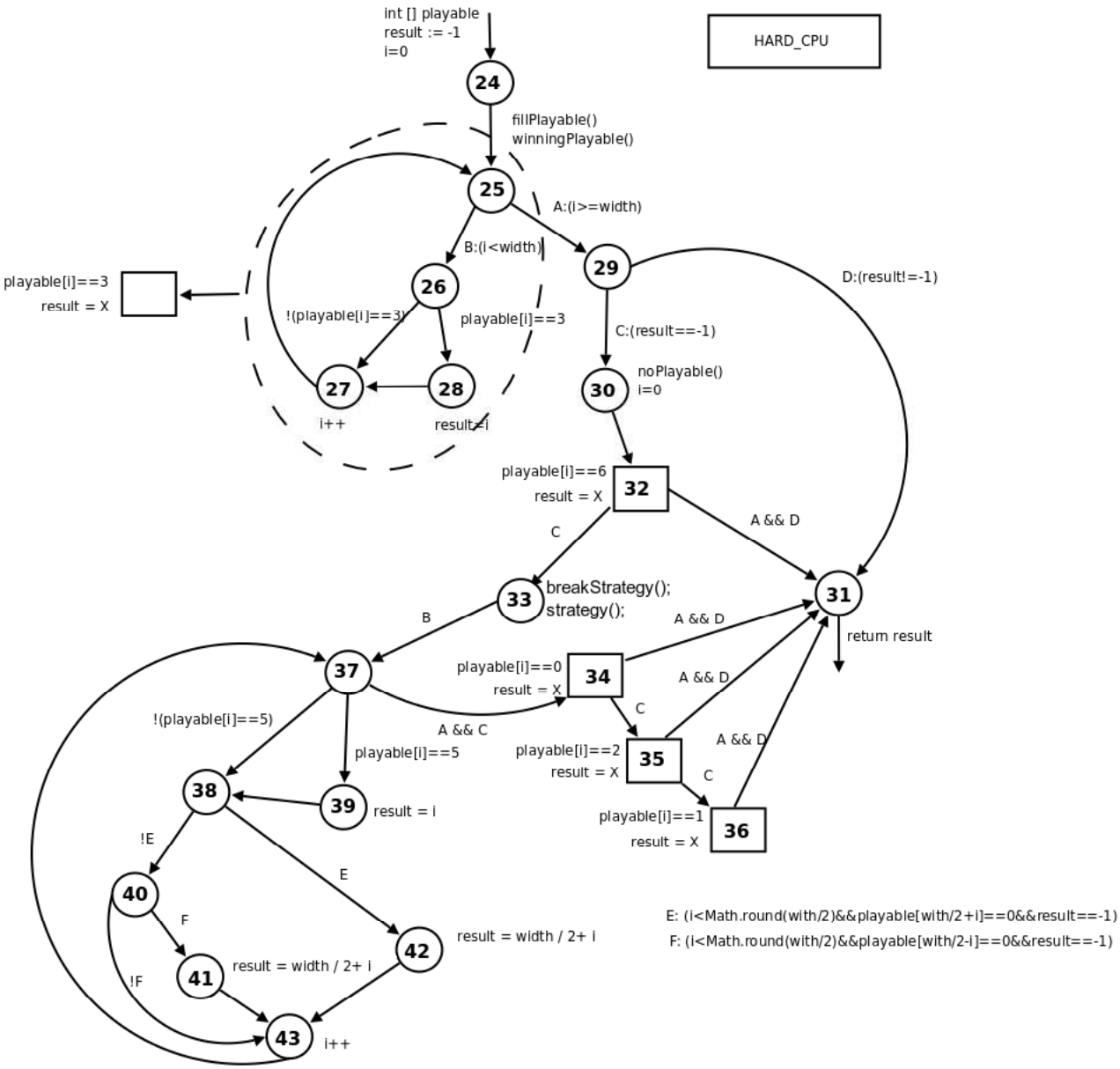

FIG. 17 – IA du programme

DT1={grille_vide}
DT2={grille_pleine_1_position_libre}

DT1={24-25-26-27-25-29-31}

DT2={24-25-26-28-27-25-29-30-32-31}

Critère de couverture : l'ensemble des sommets du graphe.
DT1 : TER1= 21%
DT2 : TER1= 31%

5.2.1 Les fonctions

`fillPlayable()`

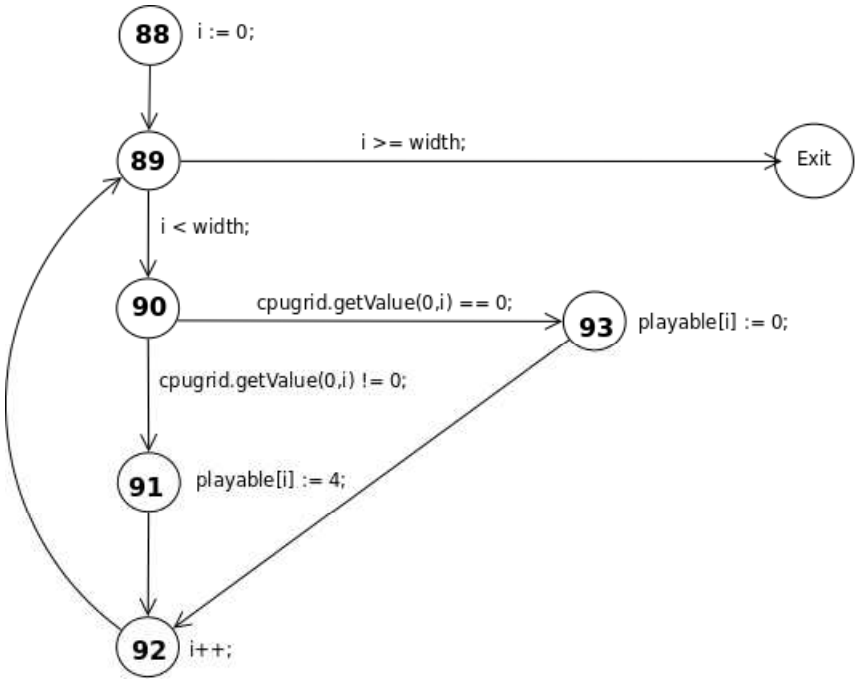

FIG. 18 – Fonction fillPlayable

DT1={grille_vide}
DT2={grille_remplie_avec_positions_libres}

DT1={88-89-90-93-92-89-...-89-exit}
DT2={88-89-90-93-92-89-...-89-90-91-92-89-exit}

Critère de couverture : l'ensemble des sommets du graphe.
DT1 : TER1= 85%
DT2 : TER1= 100%

`winningPlayable()`

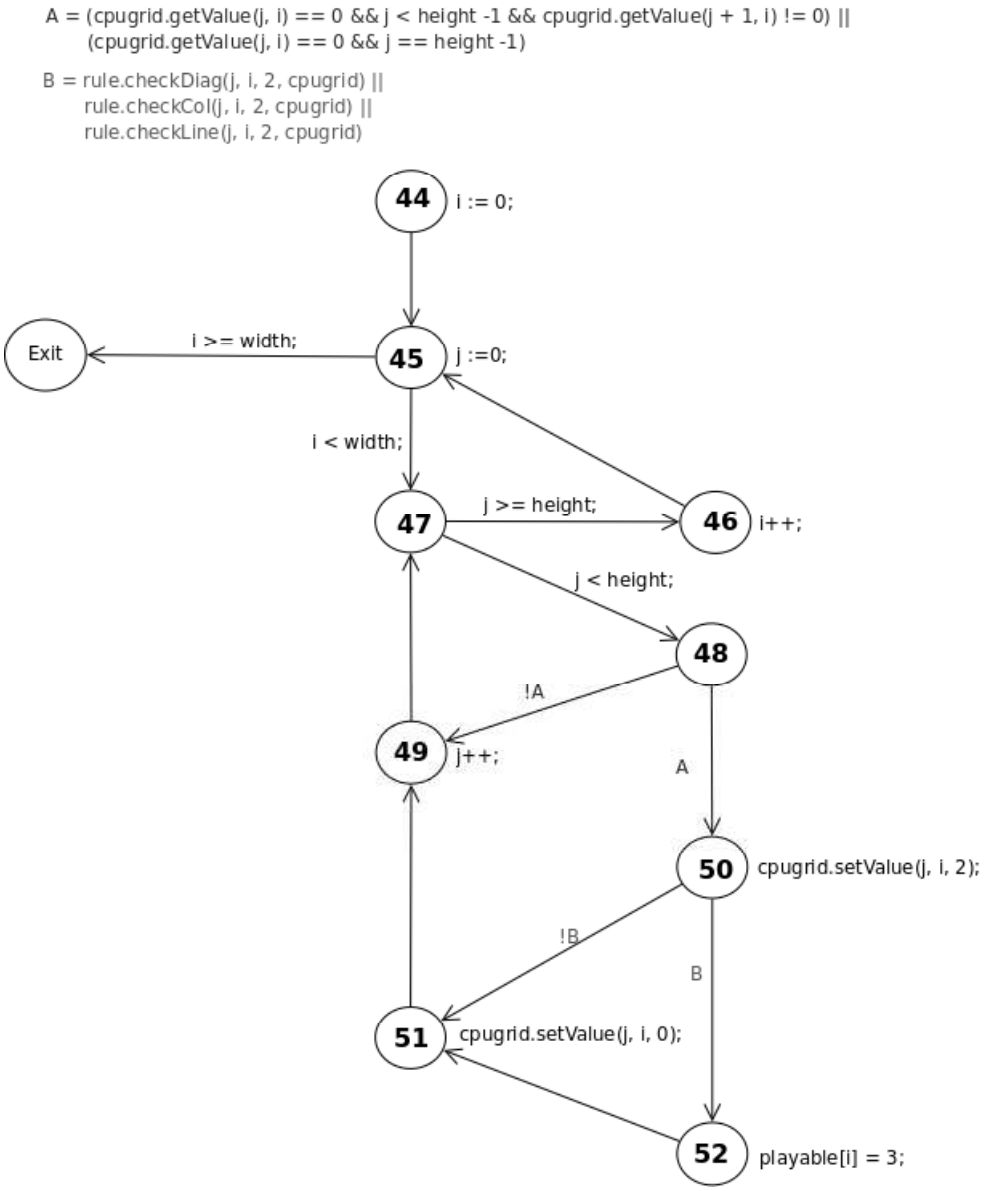

FIG. 19 – Fonction winningPlayable

DT1={grille_vide}
DT2={grille_remplie_avec_positions_libres}

DT1={44-45-47-48-50-52-51-49-47-...-46-45-exit}
DT2={44-45-47-48-49-47-...-46-45-exit}

Critère de couverture : l'ensemble des sommets du graphe.
DT1 : TER1= 100%

DT2 : TER1= 70%

Pour les 3 derniers CFA, nous n'avons pas trouvé de DT suffisament pertinentes pour être intégrées dans le rapport. Nous les avons néanmoins fournies en consultation dans les pages suivantes.

`noPlaybale()`

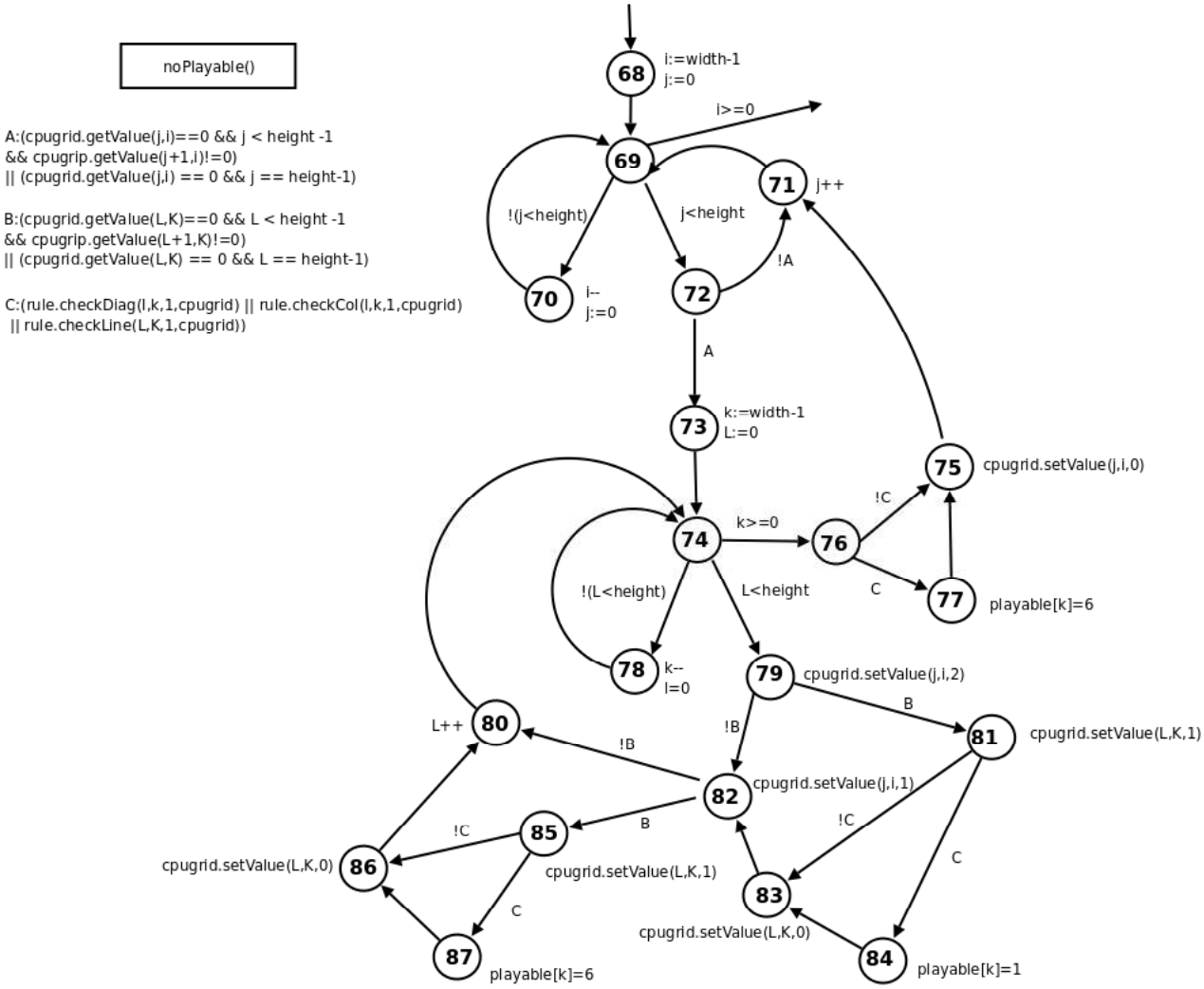

FIG. 20 – Fonction noPlayable

`breakStrategy()`

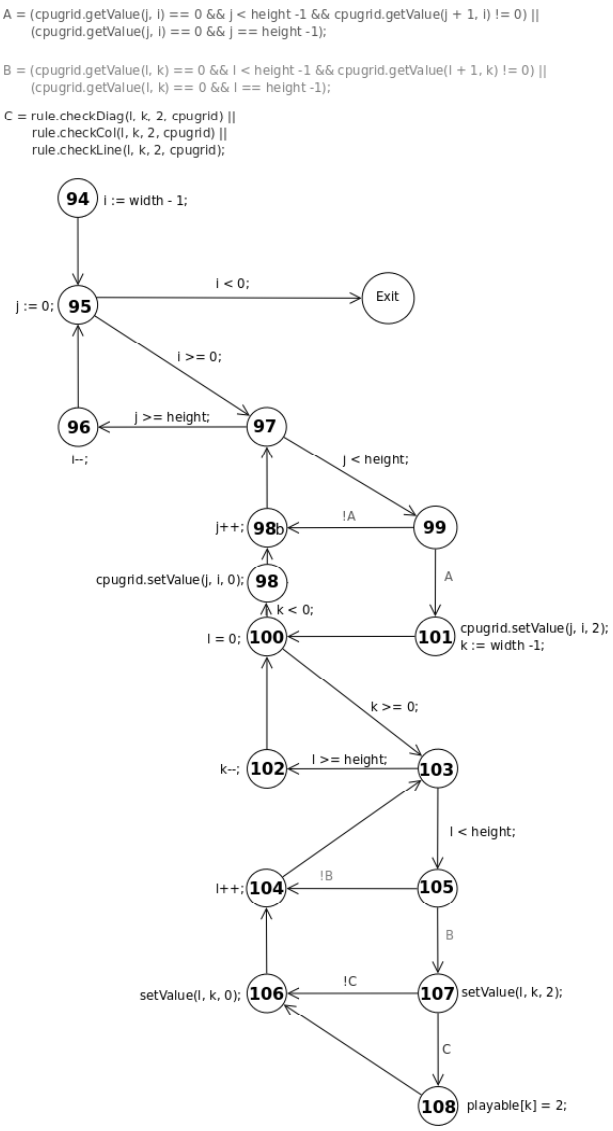

FIG. 21 – Fonction breakStrategy

`strategy()`

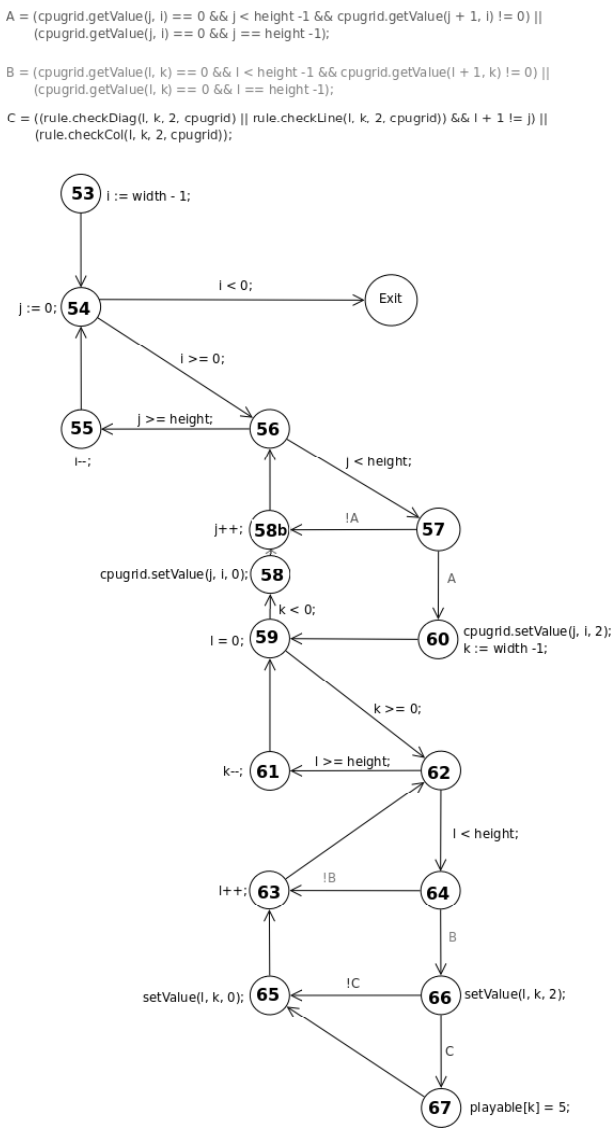

FIG. 22 – Fonction figure

5.3 Conclusion

Cette annalyse des CFA nous a permis de trouver des DT avec un pourcentage de couverture suffisant pour être pertinentes. Cela a grandement facilité la mise en place d'une stratégie de test efficace des différents modules du puissance4. Il est aussi intéressant de mentioner le fait que la création du CFA `vérification_choix()` nous a permis de corriger une erreur d'écriture de fonction. En effet nous avions codé cette fonction de telle façon qu'elle nous renvoyait toujours true, même en passant des valeurs qui sortaient des limites de la grille. Le CFA à mis en valeur ce défaut de conception et nous à permis de le corriger.

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 : existence des configurations de jeu`

  Partie Humain vs Humain : succès.
  Partie Humain vs Cpu niveau 1 : succès.
  Partie Humain vs Cpu niveau 2 : succès.

- `Test 2 : tailles de grilles`

  L'instanciation d'une grille 6x7 a été réalisée a de multiples reprises avec succès.

  Pour les autres dimensions, nous avons décidé de réaliser une analyse partitionnelle basée sur l'agrandissement et la réduction de la grille d'origine.

  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 $I_{logique} = [4, +\infty]$. On entend par valeur logique d'instanciation une valeur susceptible d'être instanciée car elle permet de jouer une partie (i.e on n'instanciera jamais une grille de dimensions -1,3 car une valeur négative n'a aucun sens et une valeur comprise entre 0 et 4 n'est pas suffisante pour qu'il soit possible de gagner). Voici donc 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'interesser 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, $I_{L1} = [4, 7]$, $I_{H2} =]7, +\infty]$.
  On détermine alors exhaustivement l'ensemble des combinaisons des classes d'équivalence :(I_{H1}, I_{L1}),(I_{H2}, I_{L1}),(I_{H1}, I_{L2}),(I_{H2}, I_{L2})

  Enfin, on choisit arbitrairement une dt par classe d'équivalence et on obtient les 4 jeux de test suivants dérivant des couples précédents : `(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 atttentes.

- `Test 3 : match nul`
  Nous avons joué une partie de manière à obtenir un match nul, un élément de l'interface graphique nous a effectivement averti de cette terminaison.

- `Test 4 : échantillon d'utilisateurs`
  L'ensemble des étudiants auquels nous avons soumis le programme ont été d'accord pour dire que les règles avaient été respectées lors de leur partie.

- `Test 5 : parties aléatoires`
  Nous avons testé avec succès la progression et terminaison correcte de dizaines de parties générées à l'aide d'un algorithme aléatoire pour le choix du coup à jouer par les 2 joueurs ordinateurs. Cette fonction est accessible en démarrant le programme avec l'option `-test`.

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 : accès aux différentes configurations`
  Chacun des boutons de l'interface graphique entraîne la création de la partie correspondante. Nous avons vérifié exhaustivement chaque bouton.

6.1.3 Types de joueurs

- `Test 7 : deux niveaux de jeu`
  De multiples parties avec les 2 niveaux de jeu ont été réalisées avec succès.

6.2 Tests d'intégration

6.2.1 Structure de données

- `Test d'instanciation correcte` Nos différents tests d'intégration sur l'instantiation de la structure de données ont réussis. On retrouve exactement les dimensions correspondant aux valeurs entrées pour chaque grille testée. Ces grilles étant inspirées de l'analyse partionnelle vue précédemment.
- `Instanciation incorrecte`
  Dans le cas où le moteur de jeu initialise la grille avec des valeurs incorrectes une grille par défaut est générée à la place. Ce test est donc passé avec succés.
- `Test de lecture/modification correcte`
  L'ensemble des tests unitaires effectué sur la structure de donnée et sur les modules joueurs (humain ou ordinateurs) permettent de déduire que les valeurs échangées grâce au moteur de jeu sont toujours correctes.
- `Test de lecture/modification incorrecte`
  Des bugs d'indices ont été mis en évidence par ce test, désormais lors d'une saisie incorrecte la lecture ou l'écriture est ignorée et un message est affiché dans le terminal.

6.2.2 Joueurs

- `Test d'instantiation`
  Un test exhaustif a montré que chaque bouton du menu permettait d'initialiser le moteur de jeu avec le bon mode et que le type de joueur correspondant était crée en conséquence.
- `Tests d'interaction`

- Cohérence des coups
  Dans la fonction *start*() du moteur de jeu, nous avons fait afficher les coups joués par les joueurs dans le terminal et avons vérifié qu'ils correspondaient aux coups choisis par le joueur. Nous avons essentiellement vérifié les résultats grâce à des tests aléatoires en mode humain vs humain.
- Coups incorrects
  Dans le cas d'un choix incorrect, comme le numéro d'une colonne pleine ou un numéro invalide, le programme ignorait ce coup et provoquait le bug suivant : un joueur jouait 2 fois successivement. Nous avons alors modifié le code de manière à ce que le moteur de jeu demande au joueur de rejouer jusqu'à ce qu'il obtienne une valeur correcte. Ce type de test est accessible avec l'option -test au lancement du programme. Le nom des boutons permettant d'y accéder est Cpu Switch.
- Intégrité de la grille

  De la même manière que pour la cohérence des coups, nous avons vérifié grâce à des tests aléatoires et combinatoires que la grille n'était pas modifiée lors d'un passage en paramètre. Nous avons utilisé la méthode d'affichage de la structure de données pour ce faire.

6.2.3 Règles du jeu

Nous avons également utilisé les modes Cpu random et Cpu Switch pour tester les actions suivantes :

- Séquences de jeu
  Test des différentes phases (voir Architecture) et de l'ordonnancement des coups.
- Intégrité de la grille
  Test d'intégrité de la grille lors du passage à `Rules`
- Condition de terminaison
  Test des différentes combinaisons de victoire ou match nul

6.2.4 Modularité

Pour ajouter un nouveau de jeu et donc une intelligence artificielle, il fallait modifier les classes `GameEngine` et `IAFourInARow` ce qui ne correspondait pas aux exigences du test. Nous avons donc du modifier l'architecture pour insérer des interfaces.

Voici un schéma représentant les zones du code concernées :

Nous avons du modifier 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. Ca a apporté une modificaiton simple et évidente sur notre `CpuPlayer`.

On trouvait donc dans le moteur jeu :

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

Qui est devenu :

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

Le changement est représenté par une classe d'implémentation`IAFourInARow` pour l'interface `Cpu`.

Une modification similaire a été apportée au module graphique.

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 experimenté sur une grille "zero" et donnera le même résultat.

Test 3 - Résultats

Tester **aux limites** la possiblité 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 IAfourInARowTest

Nous allons reprendre les exemples de la partie implémentation IaFourInRow 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 cette partie l'ensemble

des tests se fera à l'aide d'une IA en mode difficile. 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 testée. Nous avons choisi ici de ne 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` soie le même Objet que la grille `grid` passée en référence, et que le mode de difficulté soie 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());
```

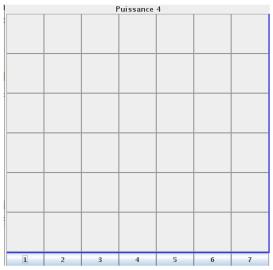

FIG. 23 – `Grille vide`

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]);
```

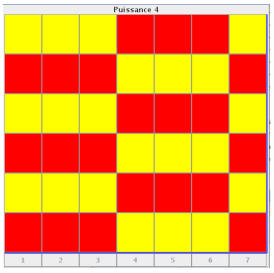

FIG. 24 – `Grille pleine`

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 ça ne se produira jamais (voir analyse statique).

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

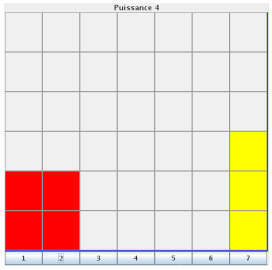

FIG. 25 – `Grille avec coup gagnant`

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]);
```

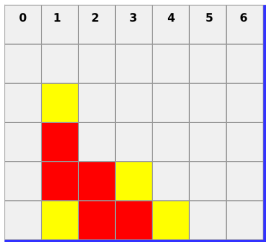

FIG. 26 – `Autre cas de coup gagnant`

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]);
```

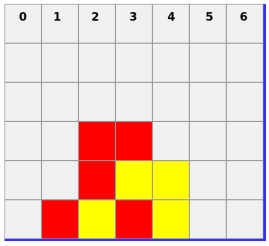

FIG. 27 – `Colonne injouable`

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]);
```

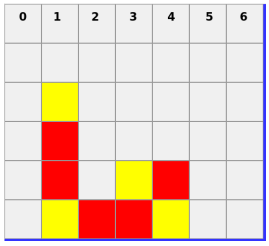

FIG. 28 – `Stratégie sur plusieur coups`

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]);
```

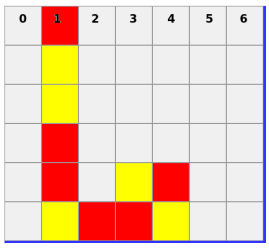

FIG. 29 – `Colonne injouable`

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 a 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]);
```

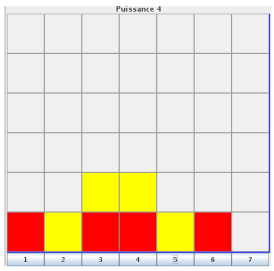

FIG. 30 – `Strategie sur plusieurs coup`

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]);
```

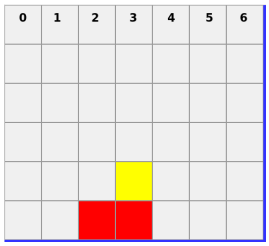

FIG. 31 – `Stratégie de blocage`

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]);
```

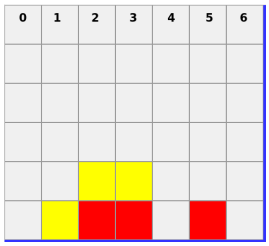

FIG. 32 – `Coup Obligatoire`

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]);
```

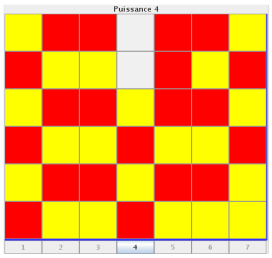

FIG. 33 – `Non blocage du jeu`

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.

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

Pour chaque état de la grille il nous était possible à l'aide de l'algorithme de connaître 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.

6.3.3 FouInRowTest

La victoire d'un joueur est conditionnée par le fait qu'un joueur aligne 4 jetons sur la même ligne, colonne ou diagonale. Ceci sera testé au moyen des 3 méthodes `testCheckLine()`, `testCheckCol()` et `testCheckDiag()`. Le résultat obtenu sera satisfaisant à la fois pour les tests d'alignement et celui de la jouabilité d'un coup donné.(cf 4.4.2)

6.3.4 GameEngineTest

Tester le moteur de jeu, nous permettra de vérifier correctement que les modes initialement initialisés seront valides. Et dans un deuxième temps, de tester les différents cas de figure de fin de partie.

Dans ce dernier point, les possibilités sont multiples et quelque unes d'entres elles seront testés(cf 4.5.2).
Mais, nous avons determiné tout de même un cas de figure non traité et impossible à tester vu l'état actuelle de notre code, donc un bug lié à notre implémentation.
C'est lorsqu'un joueur joue le dernier coup de la partie(grille remplie complètement) et aligne 4 jetons suite à ce même coup, on est incapable de désigner le vainqueur comme étant celui qui a joué en dernier.

7 conclusion

La simplicité du support Puissance 4 nous a permis de ne pas nous focaliser sur des problèmes de réalisation. Nous avons ainsi pu prendre du temps pour développer une stratégie de test ou du moins une série de tests et tenter d'aborder les différents outils et phases vus en cours. Nous nous sommes d'efforcés de couvrir une variété de tests aussi large que possible. Ainsi, on peut observer des tests classiques que des tests unitaires réalisés à l'aide de JUnit ou des parcours de chemins du programme basés sur une analyse statique. Le nombre de participants au projet n'a pas toujours facilité notre organisation puisqu'il était difficile d'organiser des réunions au travers de nos examens de Janvier. Nous estimons cependant que le code a atteint une certaine maturité et que le rapport regroupe une quantité convenable d'informations au regard du travail effectué.