Projet - Puissance 4 - Java

DUREL Enzo, VILLEPREUX Thibault

April 9, 2022

Sommaire

Introduction

1.1 Compilation

Lancez le Makefile avec la commande `make` dans le dossier `cd projet-java`.

1.2 Exécution

Exécuter le script bash **exec** qui lance le programme **Main** dans le dossier **projet-java/build**.
Si une erreur survient lors du programme, elle est écrite dans le fichier **log.txt**.

1.3 Puissance 4

Le principe est simple : insérer un pion chacun son tour dans la grille verticale. Le gagnant est le premier à avoir aligné 4 pions de la même couleur, horizontalement, verticalement, ou en diagonale. Tout l'enjeu de la partie réside dans la stratégie adoptée pour mettre l'adversaire en échec.

UML du Puissance 4

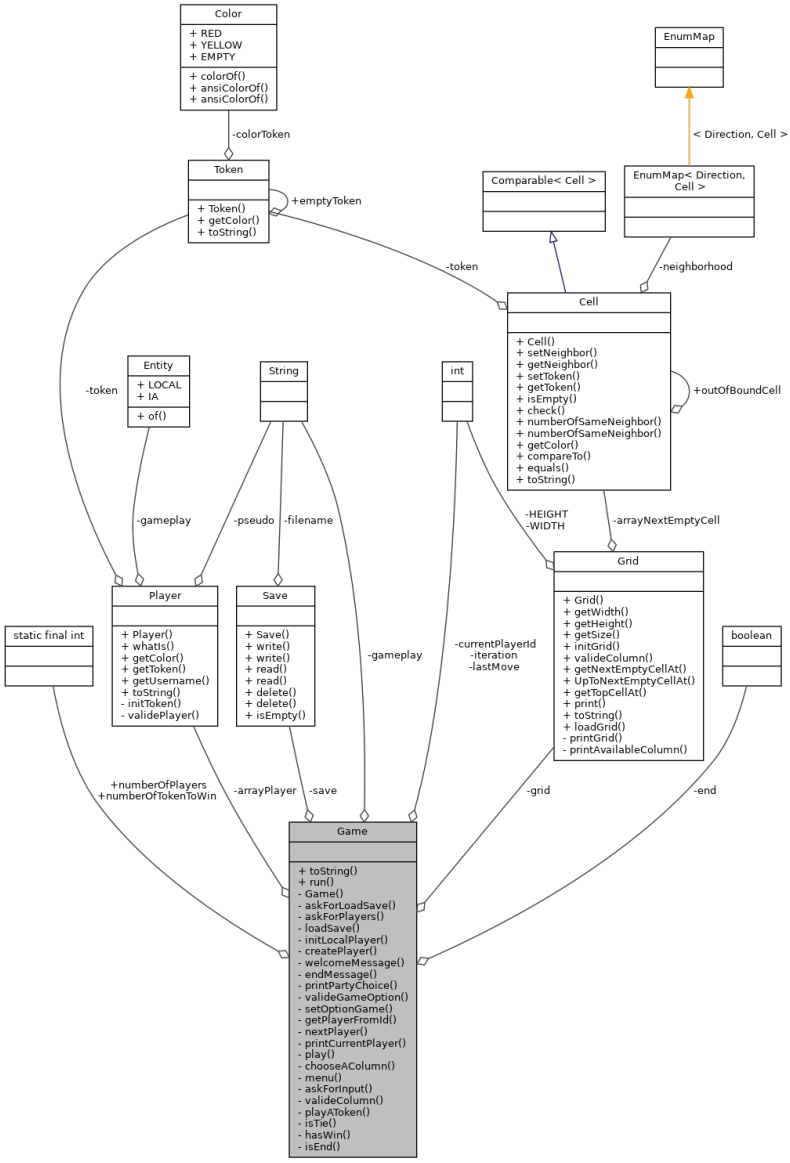

Fig. 1: UML du Puissance 4

Explication de code

3.1 Introduction

La documentation étant entièrement disponible, nous allons nous attarder uniquement sur les principales classes de notre logiciel ainsi que leur fonctionnement global.

Pour cela nous nous appuirons sur le diagramme de la classe ainsi que certaines fonctions jugées principales qui sont expliquées avec leurs graphes respectifs.

3.2 Classe Cell

Nous allons maintenant présenter la classe qui représente une **case** du plateau de jeu du puissance 4. Il s'agit de `public class Cell`.

3.2.1 Attributs

Nous avons redéfini la cellule `null`, celle-ci s'appelle outOfBoundCell et est renvoyé lorsque qu'on accès à une cellule qui n'existe pas. Nous avons rédéfini car plus compréhensible sémantiquement et pour éviter d'avoir des null qui se promène dans notre code. Cet attribut est statique donc toujours de même référence.

Chaque cellule possède un jeton (Token) qui est ici juste un objet possédant une certaine couleur (Color).

Chaque cellule possède aussi 4 voisins directes (**UP**, **DOWN**, **RIGHT**, **LEFT**) représenté ici par une EnumMap où les clés sont une direction (Direction) donnée et les valeurs la référence à sa cellule voisine dans la direction.

3.2.2 Méthodes

1. `setNeighbor: Cell*Direction -> ()`
   Cette méthode prend en paramètre une cellule (Cell) et une direction (Direction) où la cellule représente la cellule voisine a attribue dans la direction par rapport à la cellule dont la méthode est appelée.

   Elle actualise l'EnumMap représentant les voisins d'une cellule dont la clé est la direction donnée. La cellule doit être non `null`, la direction doit être non `null`.

2. `getNeighbor: Direction -> Cell`
   Cette méthode prend en paramètre une direction (Direction) et renvoie la cellule voisine dans la direction de la cellule qui appelle la méthode.

   Dans le cas où la cellule voisine est null, la cellule invalide outOfBoundCell est renvoyée. La direction doit être non nulle.

3. `check: () -> boolean`
   La fonction check vérifie si la cellule qui appelle la méthode satisfait les conditions d'arrêt du jeu: ici si 4 jeton (Token) de suite, sont de la même couleur et aligné suivant la même direction.

   Donc nous vérifions pour chaque direction possible acceptés si le nombre dans la direction donnée de jetons de la même couleurs est supérieur ou égal à 4.

   Pour cela nous utilisons deux méhodes définies en surcharges.

La première ne prend en paramètre qu'une seule direction et compte suivant les directions des cellules voisines directes (**UP**, **DOWN**, **RIGHT**, **LEFT**)

La deuxième prend en paramètre deux directions qui définiront le comptage en diagonale. En effet, une diagonale n'est juste composé de 2 directions primitives.

Les diagonales sont récupérées dans l'Enum (Direction) par la méthode `getDiagonales` et sont renvoyé sous la forme d'une EnumMap où les clés sont les directions primitives et les valeurs une direction associée formant une diagonales. Les diagonales renvoyées sont uniques.

Le nombre de voisin est définit par la somme des même cellules à partir de la cellule courante dans la direction donnée et dans sa direction opposé qui est récupéré grâce à la fonction `getOpposite` dans l'Enum (Direction).

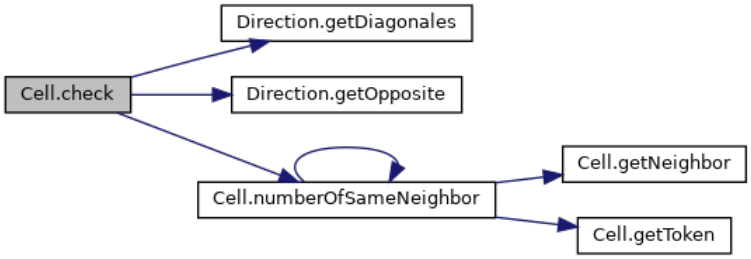

Fig. 2: Graph d'appels de Cell.check()

(a) `numberOfSameNeighbor: Direction -> int`
Cette méthode prend en paramètre une direction et compte dans cette direction et à partir de la cellule le nombre de jetons similaires.

Elle utilise pour cela la récursivité. Le cas d'arrêt est : Si la prochaine cellule est la cellule invale outOfBoundCell ou que le jeton de la cellule n'est plus le même que la cellule précédente. Sinon, elle renvoie 1 + appel de cette méthode sur la cellule suivant dans la direction donnée.

(b) `numberOfSameNeighbor: Direction*Direction -> int`
Cette méthode est similaire à la précédente, la prochaine cellule est la cellule suivant les deux directions données. La condition d'arrêt est la même, l'appel récursif est le même.

3.3 Classe Grid

Nous allons maintenant présenter la classe qui représente **le plateau de jeu** du puissance 4. Il s'agit de `public class Grid`.

3.3.1 Attributs

La classe Grid est composé d'un tableau de cellules (Cell) contenant les prochaines cellules vides de chaque colonnes. Lorsque la colonne est pleine, la prochaine cellule vide est la cellule située la plus haute dans la grille.

Nous avons utilisée cette représentation parce que sémantiquement nous n'avions besoin que de savoir cela pour permettre au joueur de jouer. De plus nous avons donc un gain de mémoire.

La classe comporte également deux attributs static représentant respectivement la largeur **WIDTH** et la hauteur **HEIGHT**.

3.3.2 Méthodes

Nous allons vous présenter maintenant quelques fonctions principales de cette classe.

1. `initGrid: () -> ()`
   Voici la fonction d'initialisation de la grille. Elle crée un tableau 2D de cellules (Cell), elle assigne à chaque cellule leurs voisins respectifs (**UP**, **DOWN**, **RIGHT**, **LEFT**) grâce à la fonction setNeighbor: Cell * Direciton ->() de (Cell).

   Nous remplissons ensuite le tableau des prochaines cellules vides définis précédement en prenant comme première cellule la cellule situé la plus en bas de chaque colonnes (car grille initialisé à vide).

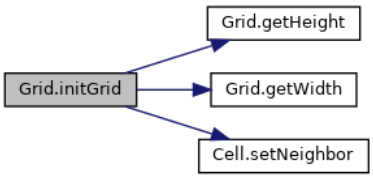

Fig. 3: Graph d'appels de Grid.initGrid()

2. `getNextEmptyCellAt: int -> Cell` Cette méthode permet de nous renvoyer la prochaine cellule à remplir pour une colonne donnée en paramètre.

   C'est un accesseur utilisé lorsque l'on veut jouer un nouveau jeton.

   Si la colonne est déjà remplie, elle nous renvoie la cellule spécifique outOfBoundCell.

   Elle vérifie quand même que la colonne existe, sinon elle renvoie une IllegalArgumentException : "column outOfBound".

3. `UpToNextEmptyCellAt: int -> ()` Cette méthode permet de mettre à jour notre tableau de cellule "arrayNextEmptyCell" avec comme indice le numéro de la colonne passé en paramètre.

Fig. 4: Graph d'appels de Grid.getNextEmptyCellAt()

Elle regarde s'il existe une cellule en haut de la dernière colonne pointée, c'est-à-dire une cellule différente de outOfBound.

Si c'est le cas, on change dans tableau la référence vers la prochaine cellule.

Cette méthode fonctionne avec effet de bord.

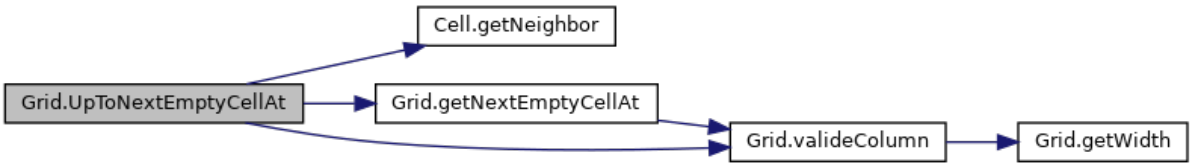

Fig. 5: Graph d'appels de Grid.UpToNextEmptyCellAt()

4. `loadGrid: String*Token[] -> ()` Cette méthode comme son nom l'indique, permet de charger la grille précédemment lue dans un fichier. Grâce à la lecture du fichier, nous obtenons un String et notre objectif est alors de le transformer en Grid.

   Pour cela on créé un tableau de cellules temporaires qui va contenir toutes les cellues de la grille sous forme de String. Remplir ce tableau se fait grâce à la fonction split(";").

   Notre tableau commence par la cellule située en haut à gauche de la grille et se termine par la cellule en bas à droite.

   Puis pour chaque cellule, on récupère sa couleur et on l'enregistre dans la grille.

3.4 Classe Player

3.4.1 Attributs

La classe comporte trois attributs :

- Le pseudo du joueur: String
- Le jeton du joueur: Token
- La nature du joueur: Entity (LOCAL ou IA)

Remarquons qu'un joueur est défini par son jeton et non pas par sa couleur.

3.4.2 Méthodes

1. `initToken: int -> Token`

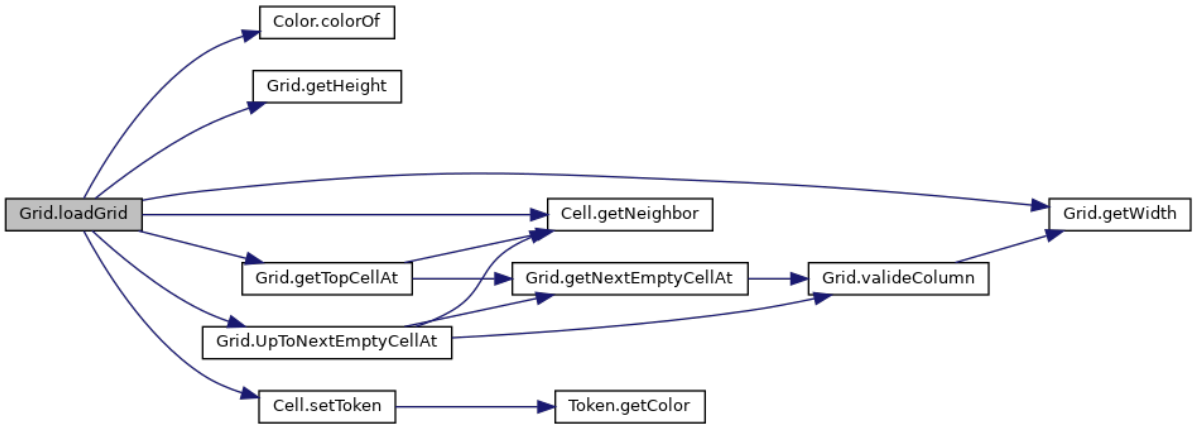

Fig. 6: Graph d'appels de Grid.loadGrid()

Cette méthode initialise la couleur du jeton par rapport à un entier donné, les entiers données sont les index du joueur dans la tableau de joueur du jeu dans la classe Game.

3.5 Classe Game

La classe représente le fonctionnement du jeu du Puissance 4, globalement elle regroupe 4 types de méthodes:

- Les méthodes d'initialisation (Joueur, Plateau).
- Les méthodes de gameplay (Choisir la colonne, Jouer le jeton, victoire).p
- Les méthodes d'affichages (Grille, joueur, menu...).
- La sauvegarde du jeu.

3.5.1 Attributs

La classe comporte 2 attributs final static :

- Le nombre de joueur au jeu : ici 2.
- Le nombre de jeton aligné pour remporter une partie: ici 4.

Les attributs de gameplay :

- Un tableau de joueur. (Un tableau car nombre fixe de joueur).
- La grille/plateau (Grid) de jeu.

Les attributs d'état du jeu:

- Un boolean indiquant si le jeu est terminé.
- Le nombre d'itération du jeu (pour tester l'égalité)
- Le mode de jeu (IA ou LOCAL)
- L'index du dernier joueur qui a joué.
- La dernière colonne jouée.

3.5.2 Méthodes

1. `loadSave: String -> ()`
   Cette méthode prend en paramètre une String qui est la représentation du contenu du fichier de sauvegarde.
   Il s'agit seulement d'une sérialisation de contenu.

2. `play: () -> ()`
   Cette méthode est la boucle principale du jeu. Elle s'arrête quand le jeu s'arrête : `this.end = true`.
   Elle exécute les actions de la boucle dans cette ordre:
   - Affichage de la grille.
   - Demande au joueur de jouer tant que son coup n'est pas valide ou

   qu'il ne quitte pas le jeu entre-temps.
   - Test si après le coup, le jeu est fini (égalié ou victoire)

3. `playAToken: Token*int -> boolean`
   Cette méthode est la méthode qui permet d'ajouter le pion dans une grille si le coup du joueur est valide.
   Elle renvoie un boolean signifiant si le coup a été correct et donc joué ou si le coup a été incorrect et donc non joué.
   Elle prend la prochaine cellule vide de la grille, regarde si cette cellule est la cellule invalide, si non, joue le jeton et actualise la dernière colonne jouée.

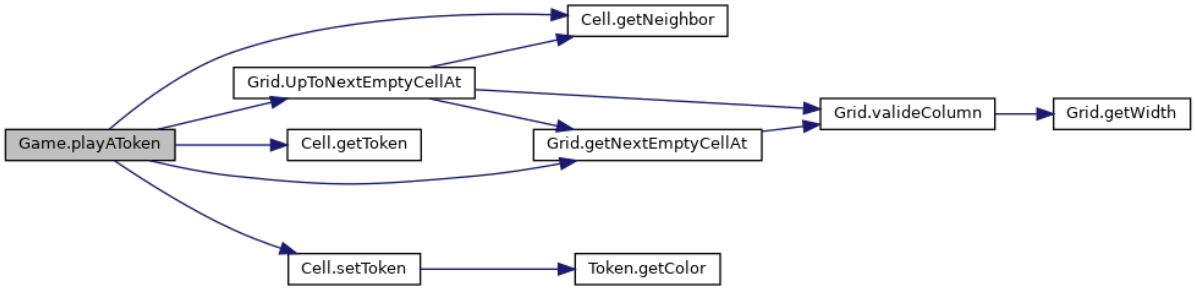

Fig. 7: Graph d'appels de Game.play()

4. `isEnd: () -> ()`
   Cette méthode teste si le jeu est terminée (victoire ou égalité), elle actualise par effet de bord l'attribut: `this.end`.

   (a) `hasWin(): () -> boolean`
   Cette méthode teste si il y a un gagnant.
   Pour cela elle appelle la méthode `check()` de la classe Cell sur la dernière cellule modifiée: càd la dernière cellules correspondant à la dernière colonne jouée.

   (b) `isTie: () -> boolean`

Cette méthode teste l'égalité du jeu: elle retourne la comparaison entre le nombre itération du jeu et le nombre de case disponible dans la grille.

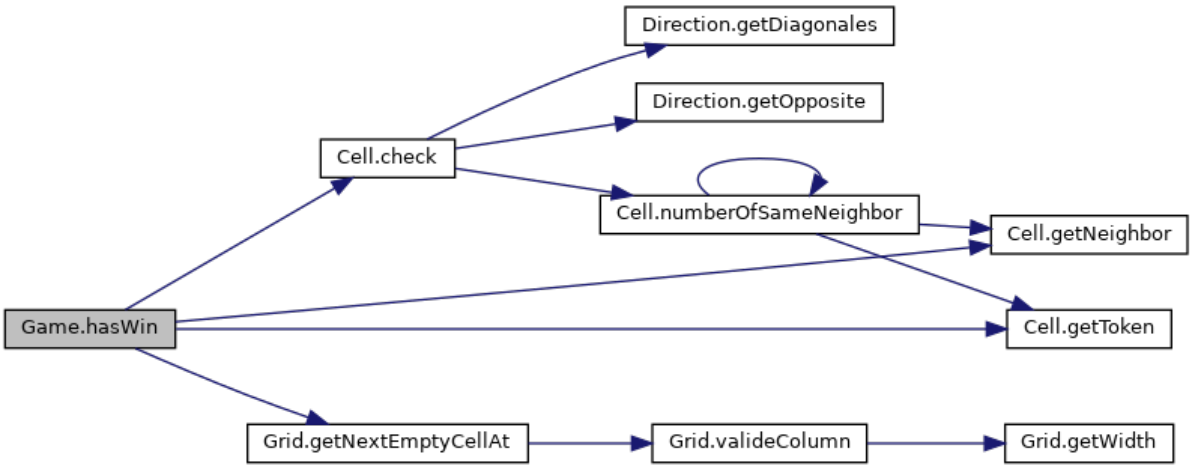

Fig. 8: Graph d'appels de Game.hasWin()

Fig. 9: Graph d'appels de Game.isTie()

3.6 Classe Save

Cette classe a été écrite pour être indépendant du jeu puissance 4, cette classe comporte juste deux méthode, lire et écrire dans un fichier.

3.6.1 Attribut

Cette classe comporte un seul attribut représentant le fichier par son chemin d'accès représenté par une String.

3.6.2 Méthodes

1. `write: Object -> ()`
   Cette méthode écrit dans le fichier l'appelle à la méthode toString() de l'objet en paramètre.

2. `read: () -> String`
   Cette méthode lit dans le fichier et renvoie sa représentation sous la forme d'une chaîne de caractères.

3.7 Enumérations

Nous avons implémenté 3 énumérations dans notre logiciel.

3.7.1 Entity

Cette énumération possède 2 attributs: **LOCAL**, **IA**. Elle définit la nature du joueur: si c'est un joueur humain (**LOCAL**) ou un ordinateur (**IA**).
Elle possède une fonction qui retourne l'Entity correspondant à une chaîne de caractères passées en paramètre: `of: String -> Entity`.

3.7.2 Color

Cette enumération possède 3 attributs : RED, YELLOW, EMPTY EMPTY désigne la couleur vide pour les cellules initialisées.
Elle possède quelques fonctions utiles comme :

- `ansiColorOf: String -> String` retourner les couleurs ansi suivant la chaine de caractères passée en paramètre.
  - `colorOf: String -> Color` retourner la couleur de type Color par une

  chaine de caractère donnée en paramètre.

3.7.3 Direction

Cette énumération possède 4 attributs qui sont les directions des cellules voisines à une cellule: **UP**, **DOWN**, **RIGHT**, **LEFT**.
Elle possède une fonction retournant les diagonales sous forme d'une EnumMap:

- `of: () -> EnumMap<Direction, Direction>`. Les clés sont les

directions primaires et les valeurs une direction formant la diagonale. La fonction est prévu pour assurer l'unicité des diagonales.

Documentation

La documentation est entièrement disponible dans le dossier **doc/**. La documentation a été générer grâce à l'outil DOxyGen (1.9.3). Elle contient le diagramme de classe de chaque classe du logiciel ainsi que les graphes d'appels (Call & Caller).
La documentation est mise sous **HTML** et **LaTeX**.
Le fichier **HTML** est disponible ../doc/html/index.html.
Le fichier **PDF** compilé à partir du **LaTeX** est disponible ../doc/latex/refman.pdf.

Git

Le projet est disponible et hébergé sur le GitLab de l'ISIMA

List of Figures