Projet - Puissance 4 - Java

DUREL Enzo, VILLEPREUX Thibault

April 8, 2022

Sommaire

1 Introduction **2**
1.1 Compilation . . . 2
1.2 Exécution . . . 2
1.3 Puissance 4 . . . 2

2 UML du Puissance 4 **3**

3 Explication de code **3**
3.1 Introduction . . . 3
3.2 Classe Cell . . . 4
3.2.1 Attributs . . . 4
3.2.2 Méthodes . . . 4
3.3 Classe Grid . . . 5
3.3.1 Attributs . . . 5
3.3.2 Méthodes . . . 6
3.4 Classe Player . . . 6
3.4.1 Attributs . . . 6
3.4.2 Méthodes . . . 6
3.5 Classe Game . . . 6
3.5.1 Attributs . . . 6
3.5.2 Méthodes . . . 6
3.6 Classe Save . . . 7
3.6.1 Attribut . . . 7
3.6.2 Méthodes . . . 7
3.7 Enumérations . . . 7
3.7.1 Color . . . 7
3.7.2 Direction . . . 7

4 Documentation **7**

5 Git **7**

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

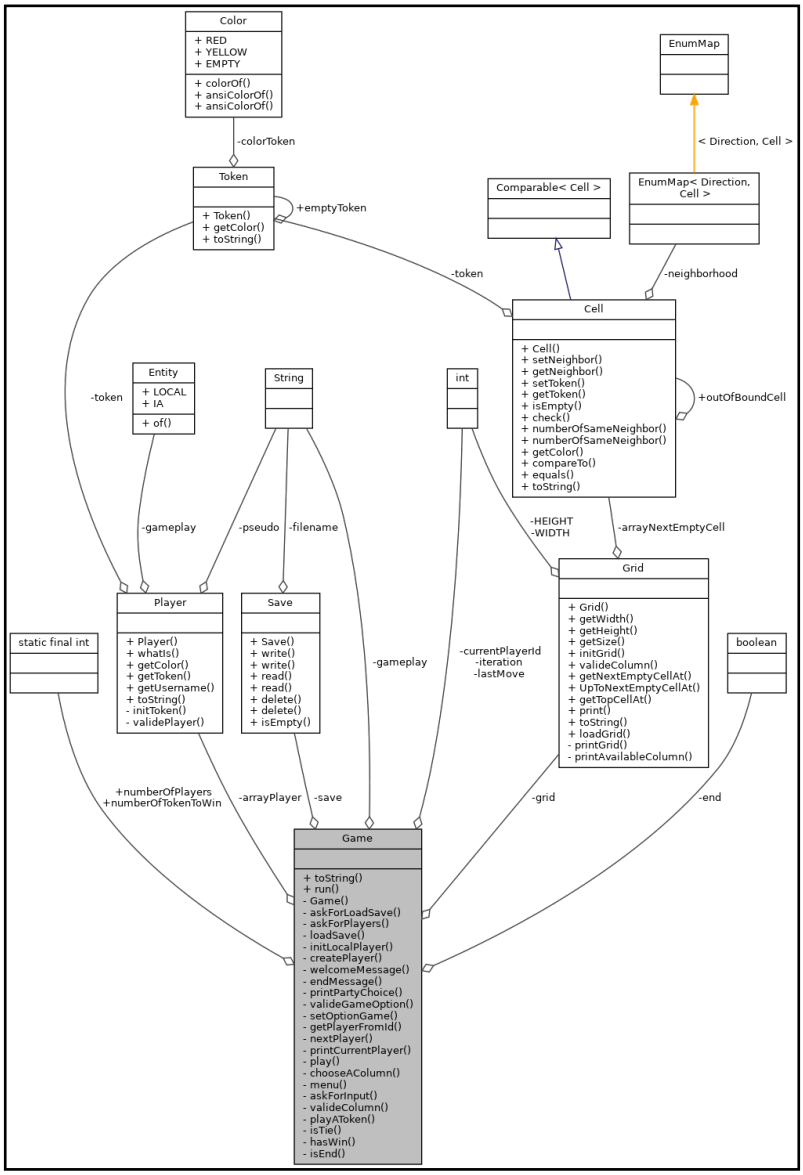

Fig. 1: UML du Puissance 4

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 **le 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).

(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).

2. `getNextEmptyCellAt: int -> ()`
3. `UpToNextEmptyCellAt: int -> ()`
4. `loadGrid: String*Token[] -> ()`

3.4 Classe Player

3.4.1 Attributs

3.4.2 Méthodes

1. `initToken: int -> Token`

3.5 Classe Game

3.5.1 Attributs

3.5.2 Méthodes

1. `loadSave: String -> ()`
2. `play: () -> ()`
3. `chooseAColumn: () -> int`
4. `playAToken: Token*int -> boolean`
5. `isEnd: () -> ()`
   (a) `hasWin(): () -> boolean`
   (b) `isTie: () -> boolean`

3.6 Classe Save

3.6.1 Attribut

3.6.2 Méthodes

1. write: Object -> ()
2. read: () -> String

3.7 Enumérations

3.7.1 Color

3.7.2 Direction

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

1 UML du Puissance 4 3