Université de Caen Normandie
UFR des Sciences
Département Informatique

3ème année de licence d'informatique

Rapport : Méthodes de Conception

Thomas Neveu (21900513)
Valentin Lucas (21901740)
Quentin Legot (21900448)

9 Décembre 2021

Table des matières

1 Patterns utilisés

1.1 Factory

Nous voulions que la façon où on construit le plateau de jeu soit facilement modulable pour pouvoir le changer facilement. C'est pourquoi nous avons opté pour le partern Factory. l'Interface contient les méthodes permettant de rentrer les paramètres de la grille qui seront appelés lors de l'initialisation de l'objet sous la forme suivante :

```
1 GridFactoryBuilder builder = LockFactoryBuilder.create().energyProbability(0.95F).
      wallProbability(0.80F).gridDimensions(12, 12);
```

> Si l'objet est mal paramétré, une erreur survient lors de la construction de l'objet car un des attributs ne serait pas initialisé.

La classe abstraite qui implémente l'interface initialise l'appel des méthodes de construction du plateau qui seront définies dans une classe qui étend de cette dernière.

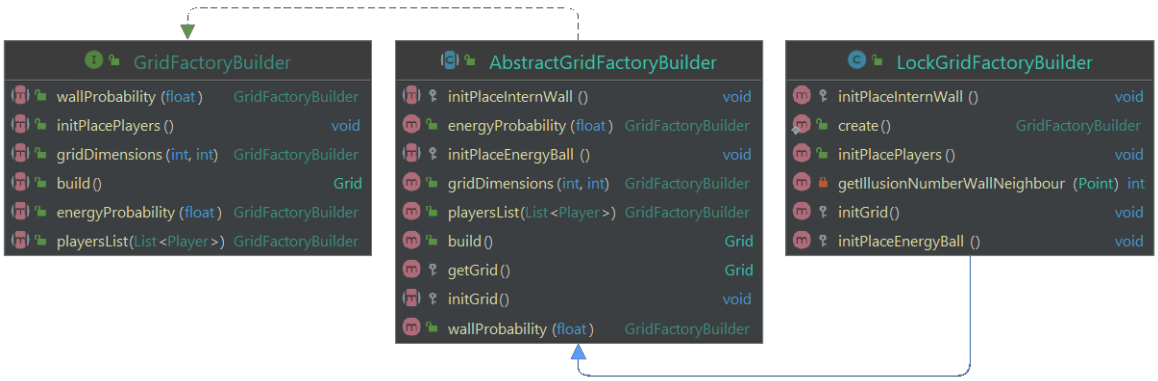

FIGURE 1.1 – UML de classe du pattern factory

1.2 Strategy

Comme pour la construction du plateau, nous voulions rendre modulable le type de joueur que nous pouvions faire jouer et donc sa stratégie de choix d'action que nous voulons lui attribuer. l'Utilisation du pattern Strategy est donc ici la bonne stratégie. l'Interface Player sert de point d'entrée dans les méthodes du joueur qui l'implémente. Un AbstractPlayer implémente les attributs communs à tout Player. Ainsi chaque classe qui étend de AbstractPlayer sera un nouveau type de joueur utilisable avec sa propre stratégie de choix d'action.

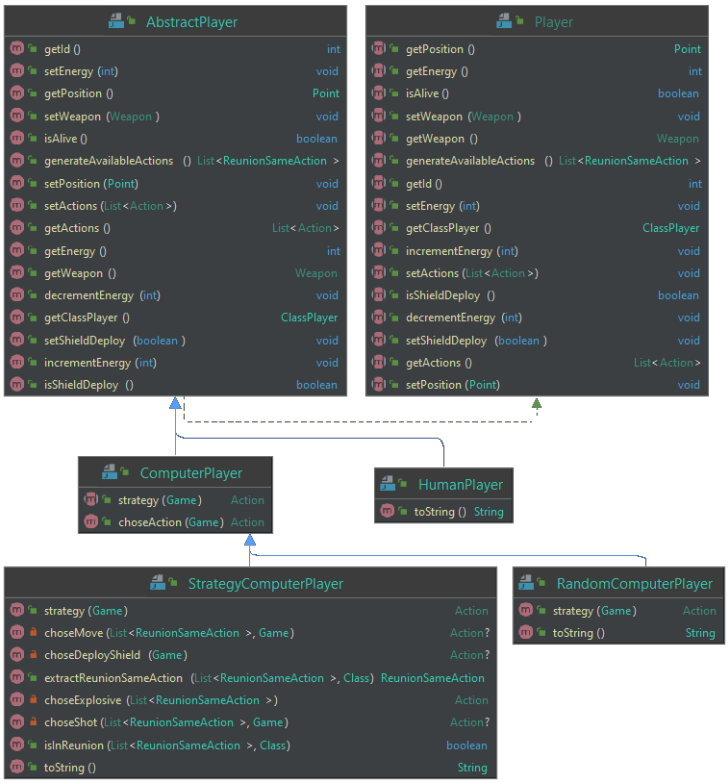

FIGURE 1.2 – UML de classe du pattern Strategy

> Ici, nous avons deux types fondamentaux de jouer, le joueur humain du type HumanPlayer et le joueur ordinateur ou IA du type Computer Player

1.3 Observer

Le pattern Observer a un lien très etroit avec le modèle MVC car il permet au modèle grâce à des listeners de savoir ses modifications. Lorsque son état change il va prévenir ses listeners grâce à un événement (ici updateModel()).

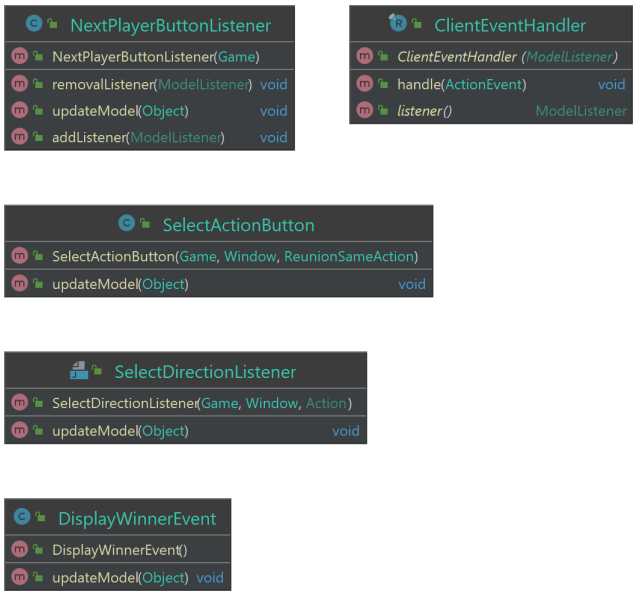

FIGURE 1.3 – UML du pattern Observer

2 Choix de conception

2.1 MVC

2.1.1 Vue

En ce qui concerne la vue, nous avons implémenté deux types de vue différentes : Une vue au niveau du terminal et une vue à partir d'une interface graphique JavaFX.
Voici un exemple de vue au niveau du terminal :

FIGURE 2.1 – Une vue terminal

Voici un exemple de vue au niveau du interface graphique JavaFX :

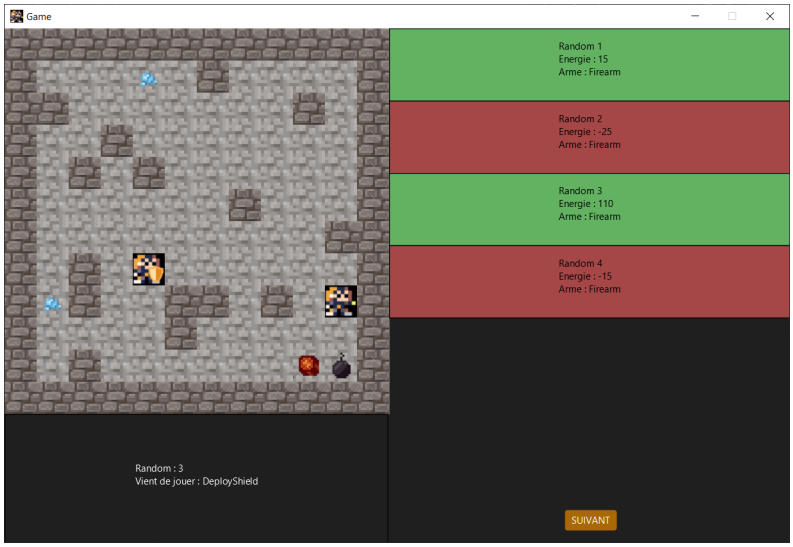

FIGURE 2.2 – Une vue window

Quelques précisions au niveau des informations visibles :

— "Random 0" signifie un joueur dont sa strategy est de type random (voir partie pattern strategy) et où son ID est de 0.

— Les cases des joueurs sont en vertes lorsqu'ils sont vivants, et passent au rouge à leur mort

> Une partie pour faire jouer les joueurs humains est aussi implémentée pour les deux modes de vue.

2.2 Gradle

Gradle est un outil moteur de production dont le but est d'automatiser les opérations répétitives comme Apache Ant.
Nous avons choisi d'utiliser Gradle plutot qu'Ant pour plusieurs raisons :

— Séparation de notre projet en 2 modules : client et server, la principale raison de cette séparation est pour éviter d'appelé les classes destiner à la vue ou au contrôleur dans le modèle. Il en est de même pour les dépendances, le module serveur ne peux pas importé de classes provenant de JavaFX.

— Téléchargement des dépendances : Gradle utilise les dépôts de Maven pour télécharger les dépendances du projet avant de les compiler et de les lancer, cela permet une plus grande modulation de notre projet, nous permettant de mettre à jour les dépendances en 2 clics grâce à nos IDEs (comme cela a été le cas avec JUnit).

— Facilité d'utilisation : même si modifier un fichier Gradle n'est pas toujours facile (de même que pour mvn ou ant), celui-ci a la particularité d'être très facile à comprendre pour quelqu'un qui ne le connais pas.