Rapport Méthode et Conception

KAMGANG KENMOE Miguel Jordan

NANMEDIGOU Yendoumban

ABOGOUNRIN Ayath

KODJO Afiwa Aimée

2 Décembre 2024

Table des matières

1 Introduction

Le projet visant à concevoir et développer un jeu de combat tour par tour consiste en la réalisation d'un jeu de stratégie opposant plusieurs joueurs sur une grille bidimensionnelle. Chaque joueur contrôle un combattant disposant d'une énergie limitée et d'un arsenal d'armes diverses. Le système intègre des mécaniques de jeu sophistiquées telles que la gestion de la visibilité partielle des éléments du jeu et un système de tour par tour dynamique.

2 Objectifs de Conception

Les objectifs principaux de ce projet s'articulent autour de plusieurs axes :

2.1 Architecture

— Implémenter une architecture MVC stricte séparant clairement la logique du jeu de l'interface utilisateur
— Concevoir un système modulaire facilitant l'ajout de nouvelles fonctionnalités
— Assurer une gestion efficace des états partiels du jeu pour chaque joueur

2.2 Patterns et Bonnes Pratiques

— Utiliser des design patterns appropriés (Factory, Strategy, Proxy) pour résoudre les problématiques spécifiques du projet
— Mettre en place un système de paramétrage flexible

3 Diagramme UML

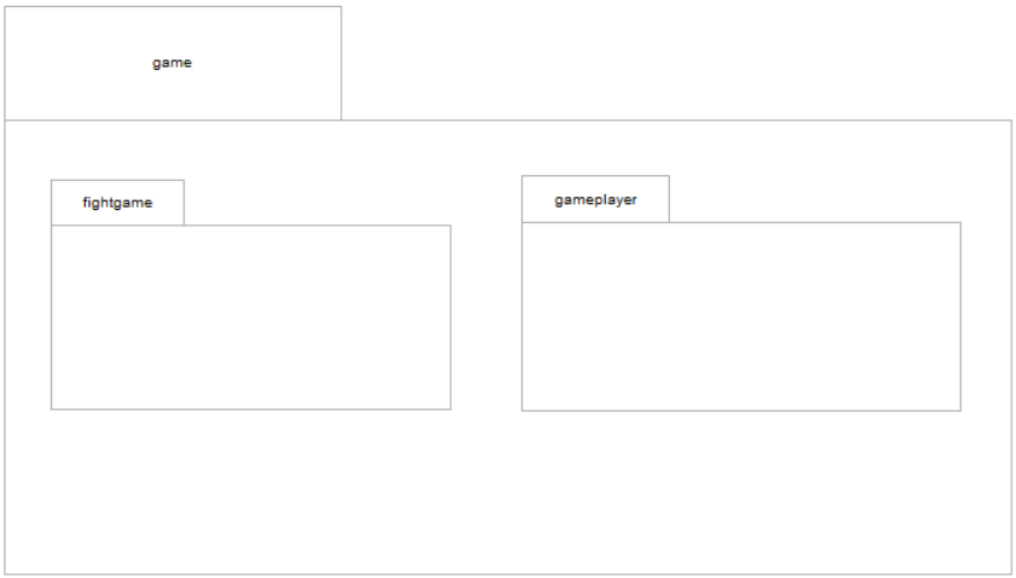

Figure 1 – Vue d'ensemble du projet

3.1 GAMEPLAYERS

Le package gameplayers contient les personnages de jeu. Il est organisé en trois sous packages.

3.2 ENTITIES

Le package entities est le principal package de gameplayers. Il est lui même organisé en trois sous-packages.

3.2.1 Package Model

Le package model contient le modèle des personnages jeu. Il contient l'interface Entity qui représente les entités d'un jeu. Une entité de jeu a une position que l'on peut récupérer. La classe AbstractEntity propose une abstraction de cette interface ; elle déclare un attribut position et une méthode pour la récupérer. Elle sera héritée directement ou indirectement par toutes les entités de notre jeu, que ce soit dans fightGame ou gamePlayers.

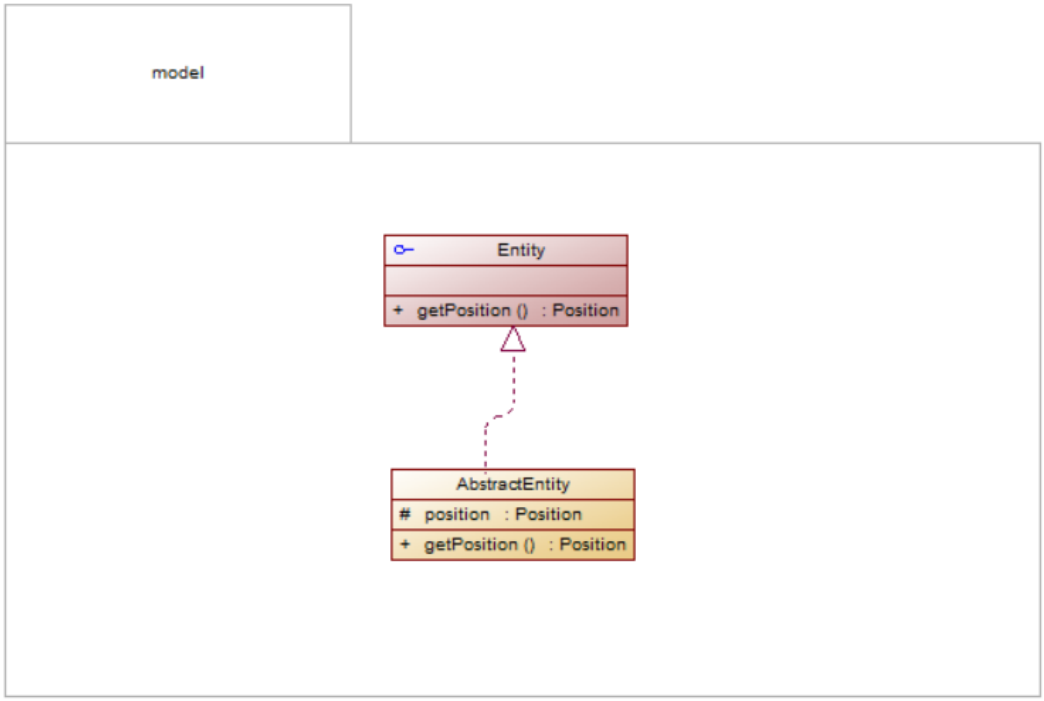

Figure 2 – Package model

3.2.2 Package Players

Le package players de entities contient les classes et interfaces permettant de représenter un joueur dans le jeu. Un joueur (classe Player) a un nom une position, des armes, un bouclier, des points de vie. Il peut être construit en prenant une PlayerConfiguration qui indique ses paramètres. De cette manière, la classe Player n'a plus à les gérer, ces derniers pouvant évoluer et devenir nombreux.

Dans la même optique, nous avons une PlayerConfigurationBuilder qui permet de créer une PlayerConfiguration en indiquant les paramètres dont on a besoin. Ainsi, pour construire un joueur, on a juste besoin du nom, de la position et de la configuration qui est par ailleurs optionnelle (une configuration par défaut sera utilisée).

L'interface Personage ajoute un niveau d'abstraction sur les joueurs que l'on peut créer. On pourra par son biais récupérer par exemple le nom, les armes ... du personnage. Un Personage peut par défaut se déplacer, tirer, activer son bouclier, planter un explosif... Aussi, la classe PlayerProxy permet de créer un proxy de joueur ; elle pourra indiquer (méthode canSee) si un joueur peut voir une autre entité ou pas.

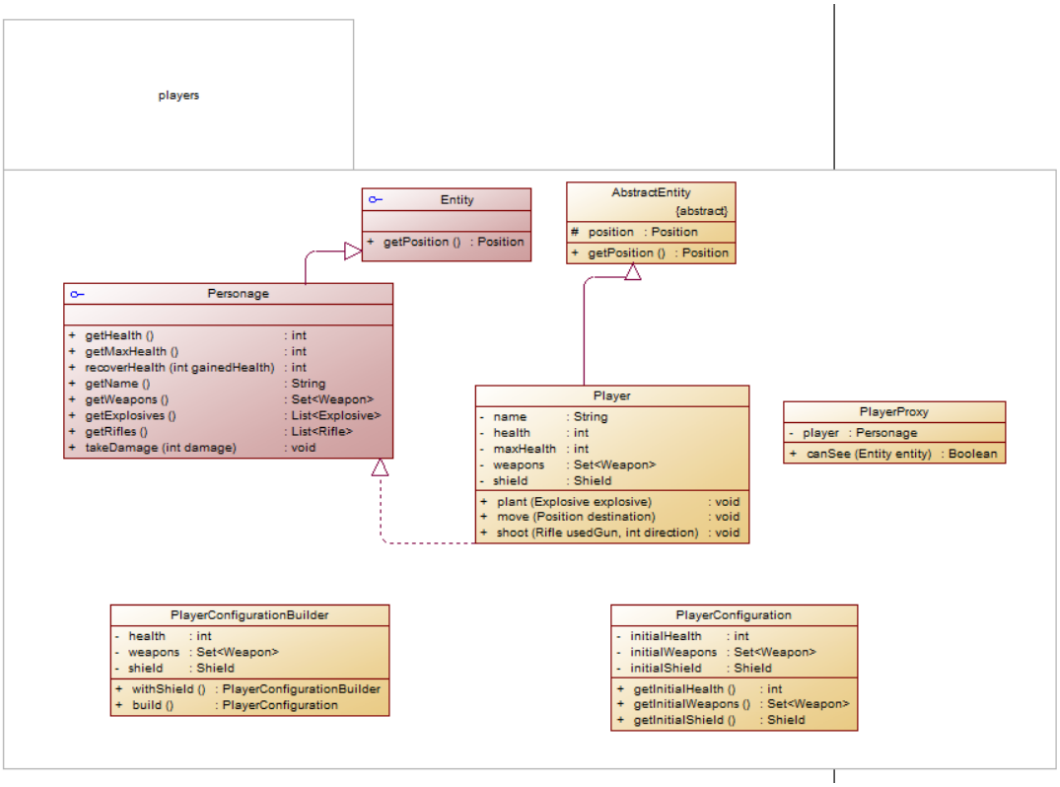

Figure 3 – Package players

3.2.3 Package Weapons

Le package weapons contient les armes des personnages. L'interface Weapon leur donne une abstraction, et nous avons la classe AbstractWeapon qui l'implémente et regroupe les éléments communs à nos armes ; une arme a un propriétaire (Personage), une puissance, et une zone de destruction (destroyedZone).

L'interface Weapon induit la définition de la méthode destroy qui remplit la destroyedZone de l'arme par les positions affectées et hit(Personage target) qui inflige des dégâts à la cible.

Nous avons deux types d'armes :

— Les fusils (classe Rifle) possèdent une portée et un nombre de munitions limité. On peut appuyer sur la gachette d'un fusil dans une direction donnée (méthode void triggerHasBeenPulled(int direction)) qui ainsi remplit sa zone de dégâts.

— Les explosifs (classe abstraite) possèdent une visibilité (attribut visible qui à false indique qu'il est seulement visible par son propriétaire) et une méthode appetizeForDestruction(Personage target) qui atteint le personnage lorsque ce dernier est posé sur lui. Nous avons pour explosifs des mines (classe Mine) et des bombes (classe Bomb, une bombe a un rayon qui indique l'étendue de sa zone de destruction et un timer).

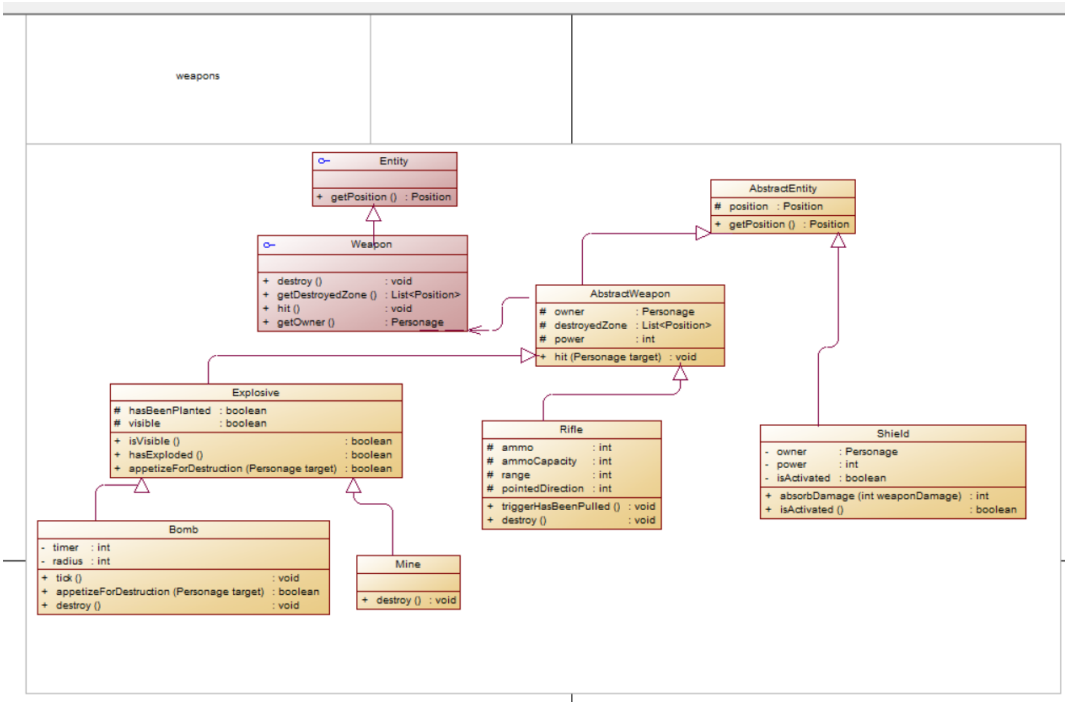

Figure 4 – Package weapons

3.3 CONFIG

Ce package contient la classe DefaultConfig à laquelle on peut demander les valeurs des paramètres par défaut des joueurs (les points de vie initial par exemple). Elle obtient ces informations en lisant le fichier de configuration par défaut.

3.4 UTIL

L'utilitaire du sous projet gamePlayers contient des classes/outils utilisés dans notre projet.

3.4.1 Util.Geometry

Le sous package util.geometry contient la classe Position qui est un élément assez central de la conception du simple fait qu'une Entity a une position. Une position a une abscisse et une ordonnée. Elle a aussi une limite en X et une une limite en Y qu'elle ne pourra pas franchir. De cette manière, une position est une position dans un espace donné. Une position peut changer en étant translater dans une direction donnée sans jamais dépasser ses limites. On peut aussi récupérer les voisins d'une positions dans son espace et dans un rayon donné. La classe Position est un véritable utilitaire de géométrie.

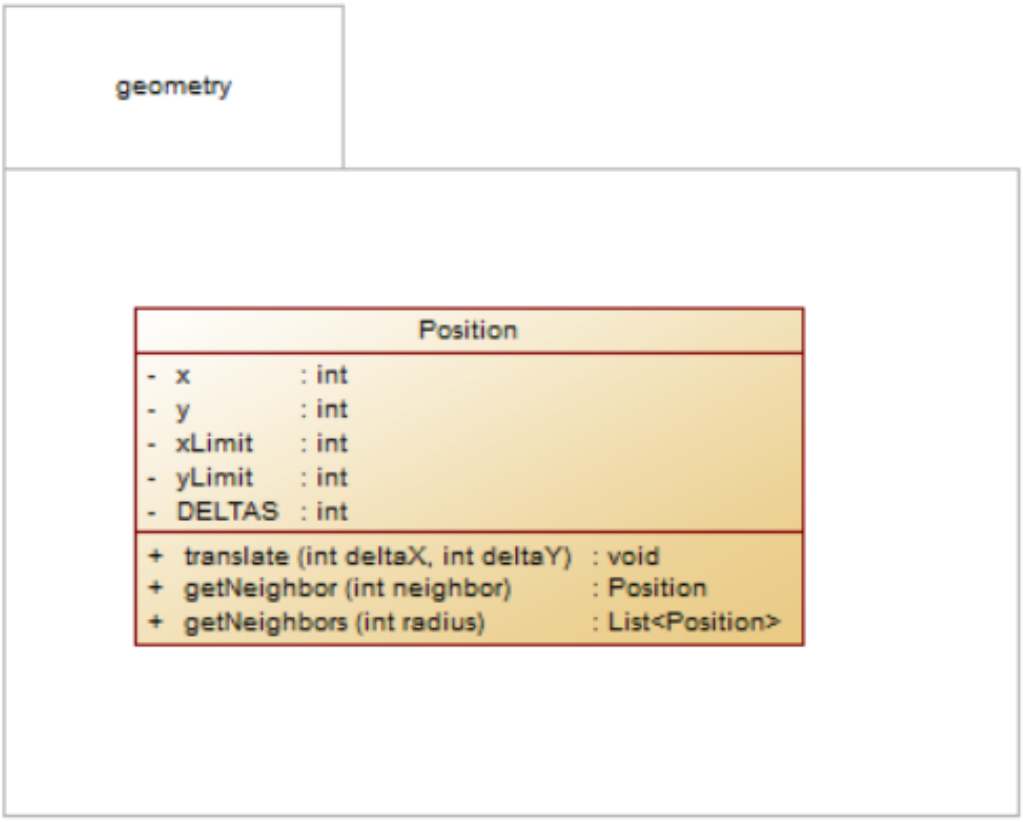

Figure 5 – Package geometry

3.4.2 Util.ConfigManagers

Le sous package util.configmanagers contient les classes permettant de lire les fichiers de configurations et d'en récupérer des données.

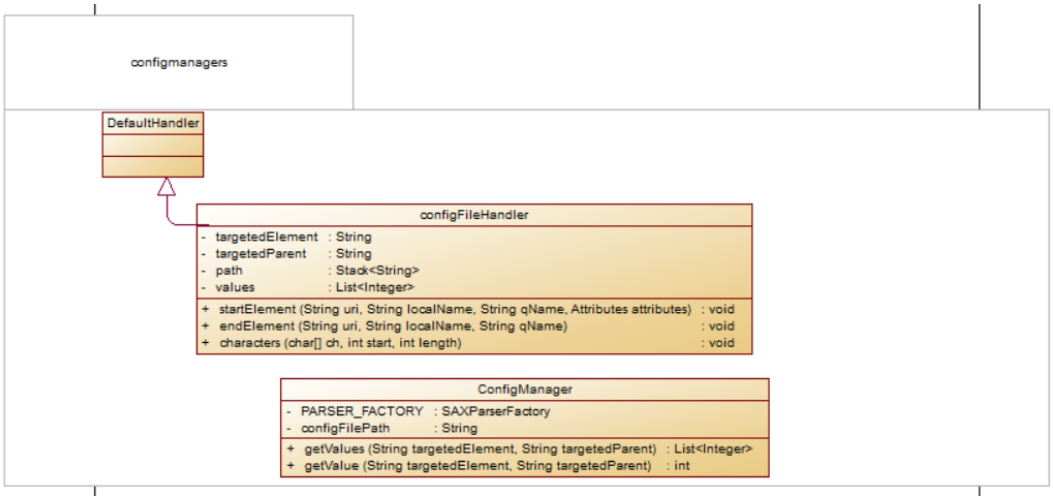

Figure 6 – Package configmanager

4 FIGHTGAME

Le projet fightGame est le jeu de combat que nous avons réalisé conformément au sujet du devoir, avec des compléments bien pensés. Il est facile à prendre en main, à réutiliser et à faire évoluer ; cela est

passé par le respect des méthodes de conception (patterns Strategy, Factory, Builder, Proxy, principe de SingleResponsability pour nos classes et méthodes...).

Conformément à la mise en place d'une architecture MVC, la partie modèle de notre jeu est complètement indépendante des interfaces graphique ou ligne commande (vue et contrôleur). Elle peut être utilisée selon d'autres modalités (via une application en réseau par exemple).

4.1 MODEL

Le package model contient le modèle du jeu. Il est organisé en sous packages cohérents représentant chacun une partie du modèle.

4.1.1 Model.Entities

Le sous-package model.entities contient les entités que nous avons créées en addition à ceux de gamePlayers. Elles sont là car elles répondent beaucoup plus à la logique du fightgame (modèle, règles) qu'à une idée de composant très génériques. Bien sûr, elles réutilisent le modèle des entités de gamePlayers (Entity et AbstractEntity ayant une position).

Il contient plusieurs sous-packages :

— entities.blocks : contient les blocs (classe Block). Un bloc est franchissable (mur) ou pas (espace).

— entities.robotsfighters : contient les joueurs robot (classe RobotFighter). Un robot à une stratégie qu'il suit et on peut récupérer sa prochaine action.

— entities.collectables : contient les objets collectables de notre jeu ; les packs de munitions (classe AmmoPack) et les pastille d'énergie (classe EnergyPellet).

Ces classes implémentent l'interface Collectable qui induit la définition de la méthode void getsCollected (Personage collecter). Pour la pastille d'énergie il s'agira d'augmenter les points de vie du joueur d'une valeur qui est un paramètre de la pastille ; et pour la pack de munitions, il s'agira de recharger le fusil le plus vide (différence entre ammo et ammoCapacity) du collecter s'il en a d'un nombre de munitions qui est un paramètre du pack. Un collectable est ramassé par un joueur qui se pose dessus dans la grille.

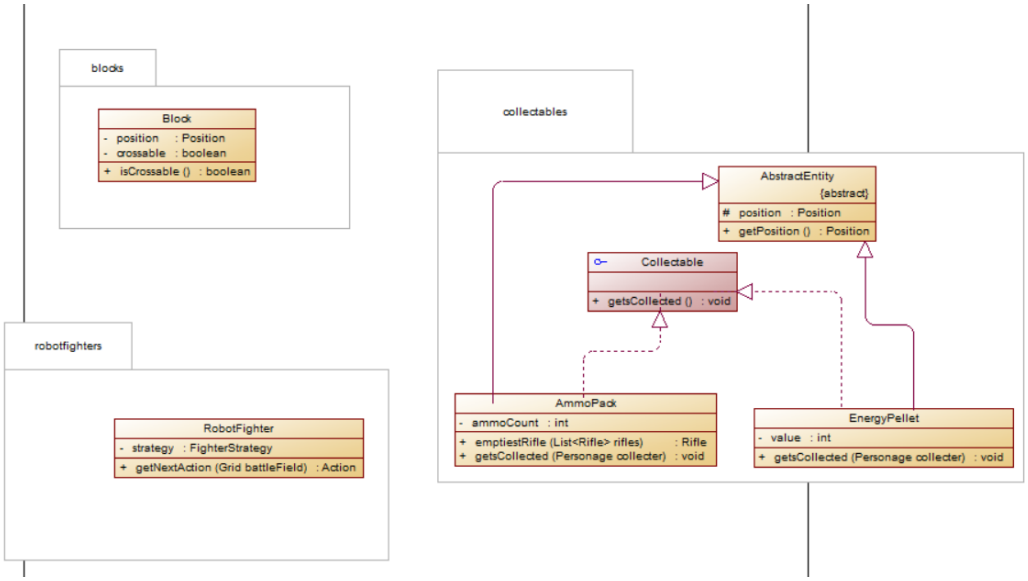

Figure 7 – Entités du package Model

4.1.2 Model.Battle

Le sous-package model.battle contient le coeur du modèle, plus précisément en la classe Battle. Cette dernière est écoutable. Une battle possède :

— Une liste de combattants (encore vivants)

— Une grille (classe Grid) qui représente le champ de bataille

— Un ensemble d'explosifs qui ont été plantés

— Une stratégie de tours (ordonnés ou aléatoires)
— Une stratégie de construction et évolution du champ de bataille

À une Battle, on peut demander si elle est terminée (boolean isOver(), true s'il reste 1 ou 0 combattants vivants). La battle évolue par le biais de sa méthode void next(Action action), donc lorsqu'une action a été réalisée. Cette méthode permettra de gérer l'action exécutée (éventuelles répercussions), mettre à jour les explosifs, les joueurs (une bombe a-t-elle explosé ? un joueur est-il mort ?...), la grid de bataille et le joueur courant.

La classe Grid qu'elle utilise pour champ de bataille contient un tableau d'Entity à deux dimensions. On pourra demander à la Grid si une cellule est vide, de nettoyer une cellule, de positionner une entité...

Pour construire une nouvelle battle on devra lui donner une setup (classe BattleSetup). Cette dernière permet la mise en place d'une battle : ajouter un combattant, définir la grid de bataille, sa stratégie d'évolution... et la validation de cette mise en place (méthode finalizeSetup). Cette méthode finalisera la mise en place de la battle (trouver les positions des joueurs en fonction de la stratégie de positionnement, les placer sur la grille) en s'assurant au préalable que la setup est valide (positions des joueurs valides en termes de leurs limites et celles de la grille, pas de conflits de positions...). Une exception est levée lorsqu'on crée une Battle avec une setup non valide.

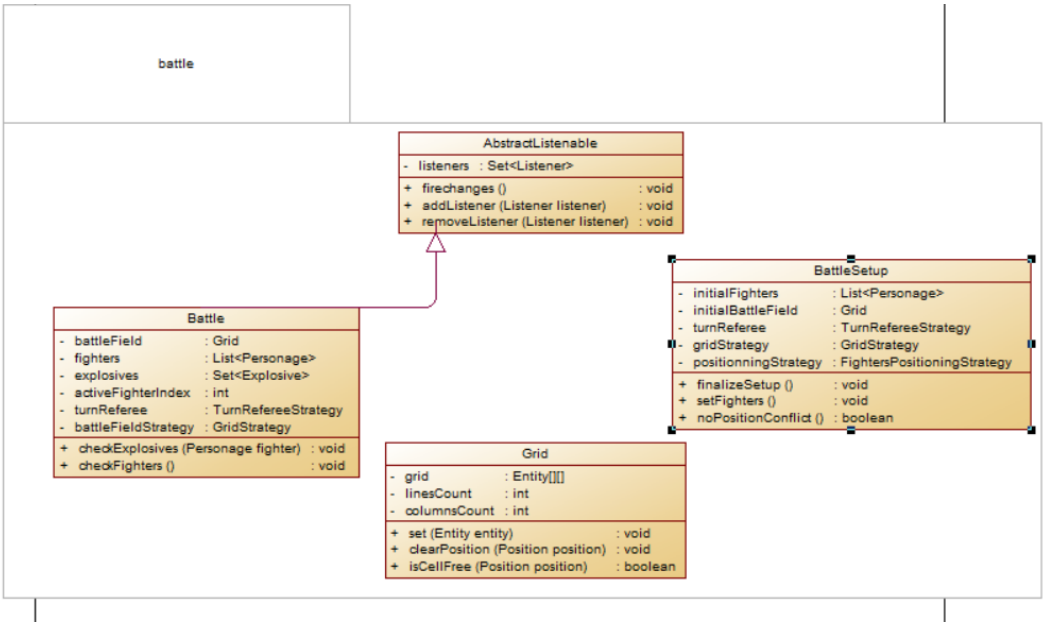

Figure 8 – Battle du package Model

4.1.3 Model.Actions

Le sous-package model.actions contient les actions que peuvent effectuer les joueurs conformément aux règles du jeu et permettant de faire évoluer une battle. Nos actions sont :

— Le déplacement (classe Moving)
— Le tir (classe Shooting)
— Le plantage d'explosif (classe Planting)
— Le levage de bouclier (classe Shielding)
— Le repos (classe Resting)

Toutes ces classes implémentent l'interface Action qui induit la définition de getsExecuted(). Elles le font en héritant AbstractAction qui regroupe les comportements similaires d'une action (possession d'un Personage performer). Pour certaines actions, il faudra indiquer d'autres paramètres (direction de tir et fusil utilisé pour le Shooting par exemple).

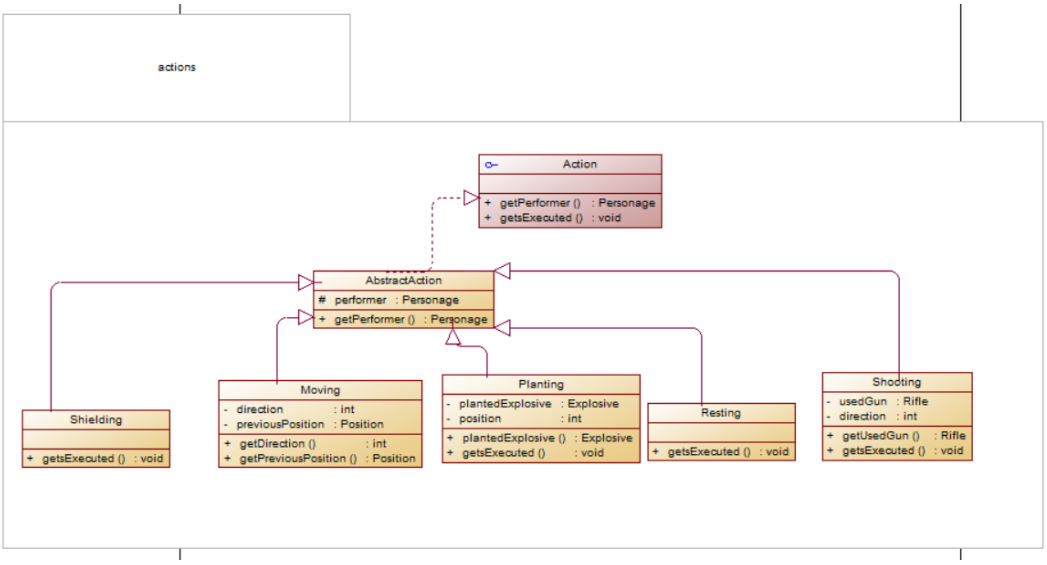

Figure 9 – Actions du package Model

4.1.4 Model.Factories

Le sous-package model.factories contient la classe FighterFactory qui nous permet de créer des joueurs avec des configurations différentes. Nous avons les types de combattants suivants :

— Demolisher : beaucoup d'armes, spécialité explosifs, peu de points de vie
— Sniper : beaucoup d'armes, spécialité fusils, peu de points de vie
— Tactician : tous paramètres moyens
— Defender : peu d'armes, beaucoup de points de vie
— Tanker : très peu d'armes, énormément de points de vie

La création se fait simplement : maFactory.createFighter(fighter, FighterFactory.DEMOLISHER) ou maFactory.createFighter(nom, position, FighterFactory.DEMOLISHER). La FighterFactory utilise la configuration par défaut ; cependant on peut lui donner un autre ConfigManager ou un autre fichier de config et il créera les joueurs avec les configs spécifiés.

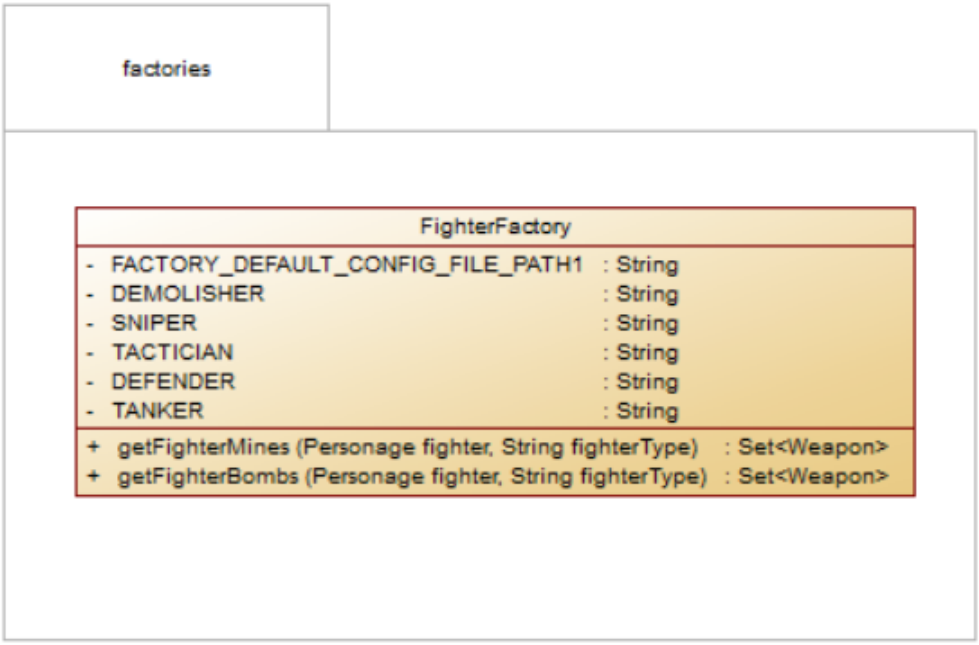

Figure 10 – Factories du package Model

4.1.5 Model.Strategies

Le sous-package model.strategies contient les stratégies de notre jeu. Elles sont toutes construites selon le pattern Strategy.

Stratégies des joueurs robots Dans strategies.fighterssrategies : L'interface FighterStrategy leur donne un niveau d'abstraction à elles toutes, qui plus est elles héritent de AbstractFighterStrategy qui contient des méthodes utilisables par toutes nos stratégies (savoir si telle action est applicable). Une stratégie définit une méthode Action computeBestMove(Personage robot,Grid battleField).

Nous avons commencé par créer une stratégie qui choisit l'action à exécuter au hasard (classe ChaoticStrategy). Puis nous avons créé des stratégies plus fines. La classe abstraite TacticalStrategy permet de savoir si un ennemi est sur la même ligne ou même colonne que le bénéficieur de la stratégie, si un ennemi est dans le périmètre du bénéficieur dans un rayon donné.

La première TacticalStrategy est l'AttackerStrategy : si le joueur a un ennemi sur la même ligne ou colonne, il tire ; s'il y'a un ennemi dans son périmètre, il plante un explosif avec un timer de 2 pour les bombes pour que ce dernier n'ait pas le temps de s'enfuir. La deuxième est la DefenderStrategy : se déplacer dans les directions inverses pour éviter les tirs en cas d'ennemi sur la même ligne ou colonne, lever le bouclier en cas d'ennemi dans le périmètre.

Notons que la TacticalStrategy a une fall back strategy (qui n'est autre que la ChaoticStrategy) qui est suivie lorsqu'elles ont incapables de déterminer l'action optimale. D'autres stratégies pourront être créées en prenant la Battle entière et non seulement la grille, et donc en étant plus informées.

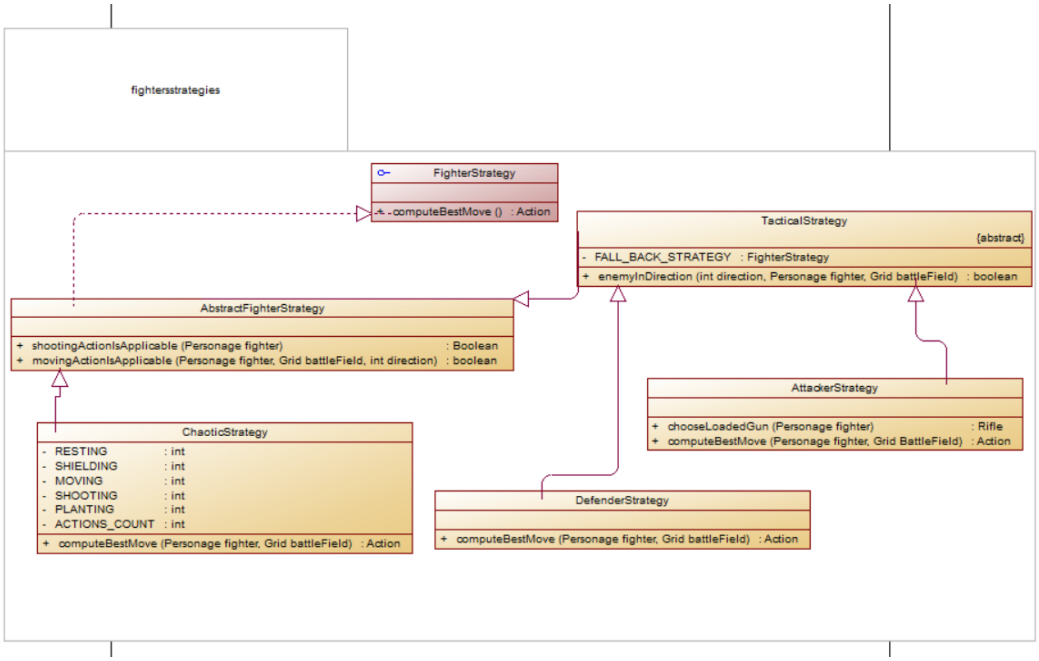

Figure 11 – Stratégies des joueurs robots

Stratégies de remplissage des grilles Dans strategies.gridstrategies : Nous avons l'interface GridStrategy et la classe AbstractGridStrategy qui abstraient nos stratégies. Cette dernière définit des méthodes permettant de trouver une cellule vide dans la grille ou de compter les collectables par exemple, méthodes partagées par toutes les éventuelles stratégies implémentées.

Une stratégie de grille permettra d'initialiser une grille (méthode initializeGrid(Grid grid)) ou de la faire évoluer (méthode evolveGrid(Grid grid)). Nous avons implémenté une RandomGridStrategy qui initialise une grille en mettant des murs et des collectables aléatoirement sur la grille en proportions définies. Pour ce qui est de l'évolution de la grille, lorsqu'un collectable a été ramassé, un autre est créer avec 1 chance sur 2.

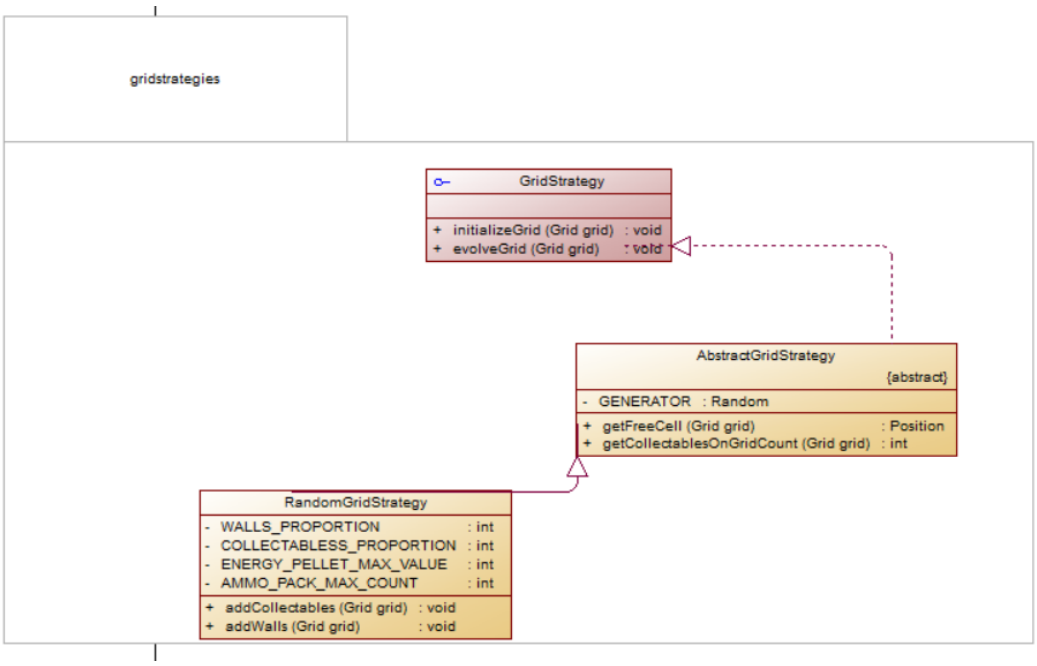

Figure 12 – Stratégies de remplissage des grilles

Stratégies de positionnement des joueurs Dans strategies.fighterspositioningstrategies : À une FighterPositioningStrategy (interface), on peut demander une liste de positions que celle-ci calcule en lui donnant le nombre de joueurs (donc nombre de positions demandées), le nombre de lignes et de colonnes de la grille.

Nous avons deux stratégies de positionnement :

— La RandomPositionning qui choisit des positions aléatoires pour les joueurs

— La SafePositonning qui les place si possible éloignés les uns des autres et pas sur les mêmes lignes et colonnes. Cette dernière stratégie utilise un algo similaire à celui du problème classique des nReines.

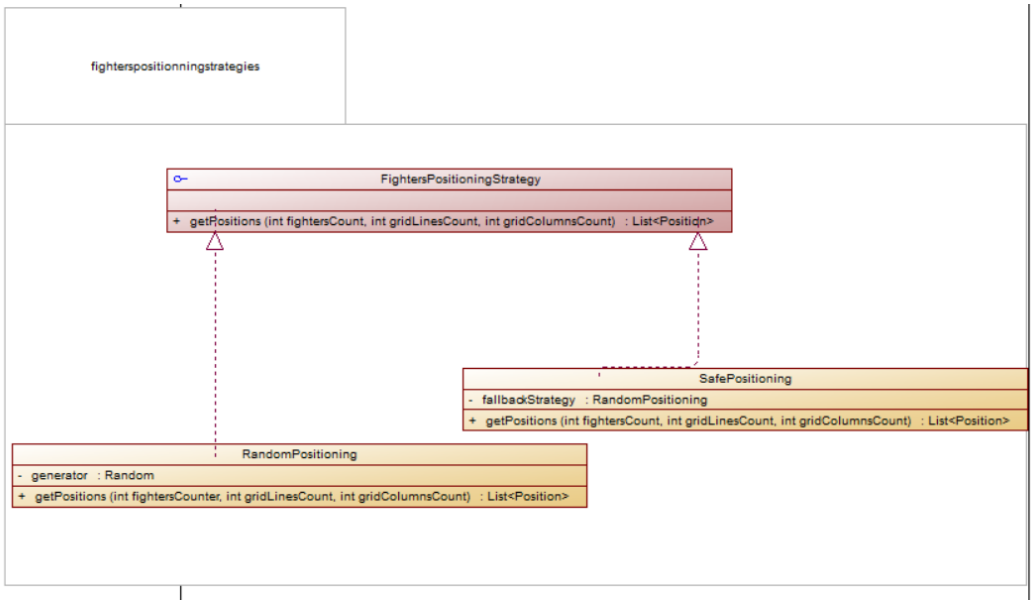

Figure 13 – Stratégies de positionnement des joueurs

Stratégies de tours de jeu Dans strategies.refereestrategies : Il s'agira d'indiquer si les tours de jeu seront choisis aléatoirement (RandomTurnsReferee) ou de manière définie (OrderedTurnsReferee).

D'autres stratégies pourront être créées en se basant sur les scores par exemple.

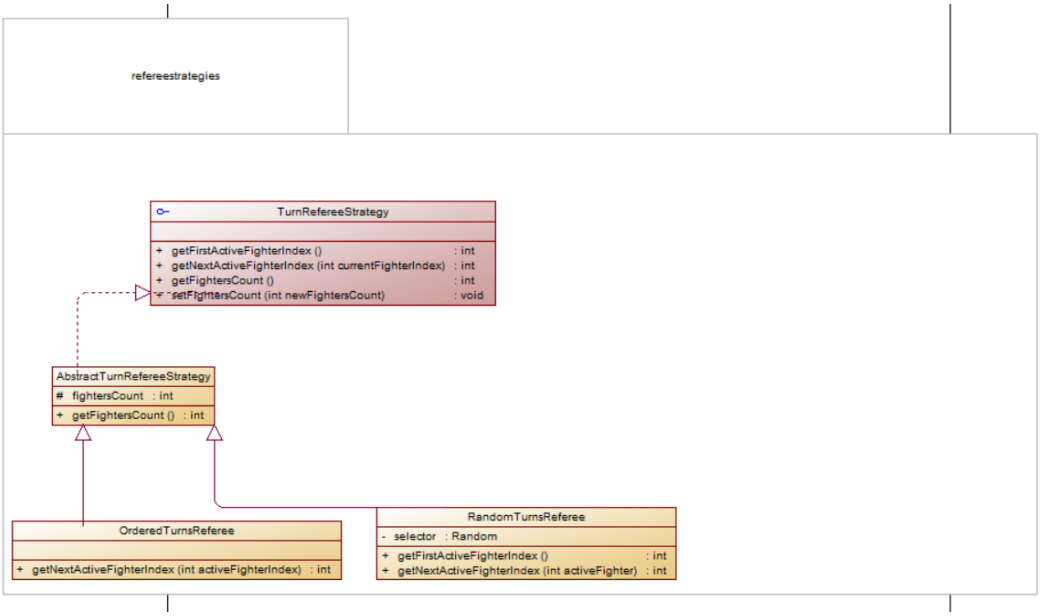

Figure 14 – Stratégies de tours de jeu

5 VUE ET CONTROLEUR

Nous avons réussi à bien séparer notre vue (package view) de notre contrôleur (package controller). Le contrôleur se charge de récupérer les choix de l'utilisateur/joueur (keyevents ou boutons pour l'interface graphique, saisies sur terminal pour l'interface en ligne de commande) et d'ainsi faire avancer le modèle (La Battle). La vue (qui est écouteur) se chargera juste faire et mettre à jour les affichages. Il convient toutefois de discuter des deux dans une même section.

Nous avons deux interfaces :

— Sur le terminal (dans les packages view.cli et controller.cli, cli pour Command Line Interface)

— Sur interface graphique swing (dans les packages view.gui et controller.gui, gui pour Graphic User Interface)

5.1 CLI

Nous disposons d'une interface en ligne de commande agréable et facile à prendre en main.

La classe Prompter du contrôleur permet de récupérer les entrées du joueur. La classe BattleConfigurator permet de configurer une battle avant de la lancer ; c'est en quelque sorte le menu principal. La classe BattleSimulator se charge quant à elle de simuler une bataille déjà configurée (demande l'action du joueur actif tant que la bataille n'est pas terminée). La classe App (fightgame.controller.App) lance la fenêtre principale du jeu. C'est elle qu'il faut exécuter pour jouer.

Pour ce qui est de la vue, nous avons la classe MenuDisplayer permet de faire les affichages relatif au menu principal et à la configuration des battles. Elle est utilisée par le contrôleur BattleConfigurator. La classe BattleDisplayer permet de faire les affichages relatif à la bataille (affichage de la grille, du joueur courant...). Elle écoute la Battle et met à jour automatiquement l'affichage à jour lorsque le modèle évolue, conformément à la conception MVC. Elle est utilisée par le contrôleur BattleSimulator.

N.B La vue n'agit jamais sur le modèle, elle se contente de le lire ; seul le contrôleur fait évoluer le modèle.

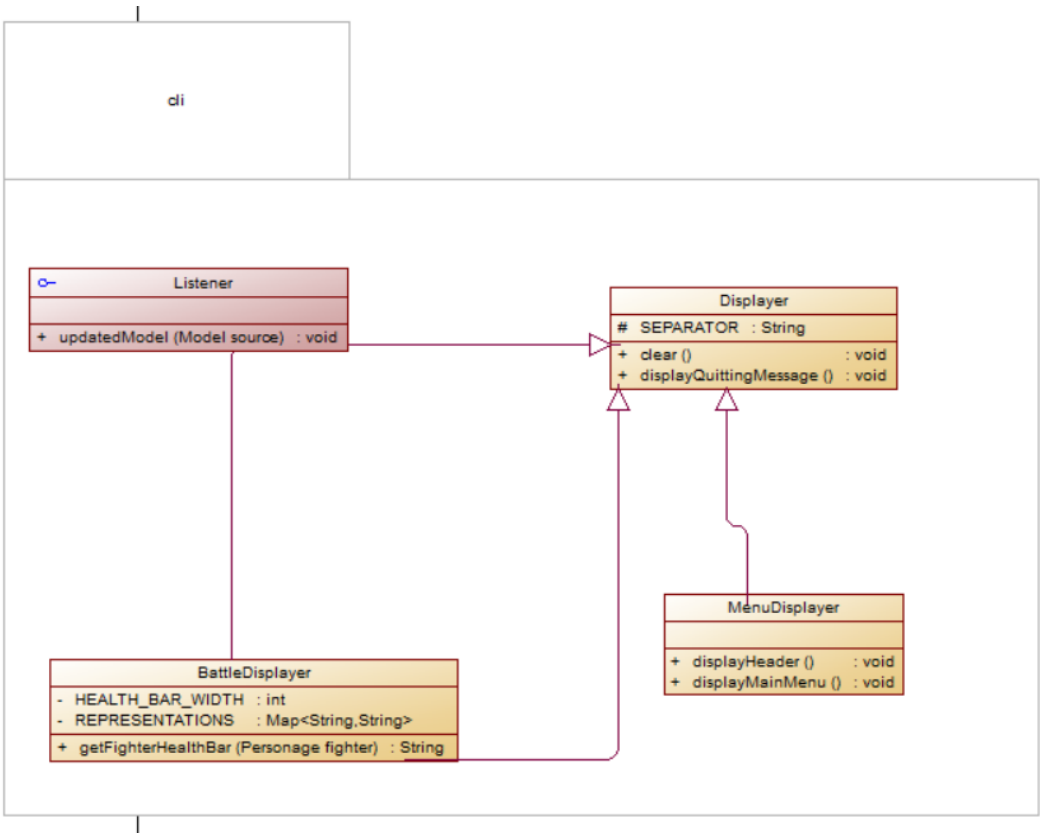

Figure 15 – Interface en ligne de commande

5.2 GUI

Pour la partie Graphique nous avons decidé de separer la Vue du Contrôleur

1. Contrôleur

— Le controleur est utiliser par la vue et prends les actions faites dans la vue, le passe au model qui va ensuite l'executer.

2. Vue

— Dans la vue nous avons un GamePanel qui ecoute le model. Il est utilisé pour representer la grille du model. Il est utilisé par la vue des personnages (RobotView.java, PlayerView.java) et aussi par la vue global BattleView.java

— La classe PlayerInfoPanel est utilisé pour afficher les armes d'un personnage en temps réel

— Nous avons aussi la classe LogPanel qui elle affiche tous les actions effectués par les joueur dans le BattleView.java

— Notre classe Menu permet de choisir les joueurs (au moins deux) et aussi une configuration du jeu (sinon la configuration par defaut est utilisée). Apres cela elle affiche l'ecran la vue global du jeu et les ecrans de chaque joueur

— Notre classe App permet d'afficher le menu

Figure 16 – Menu sans ajout de joueur

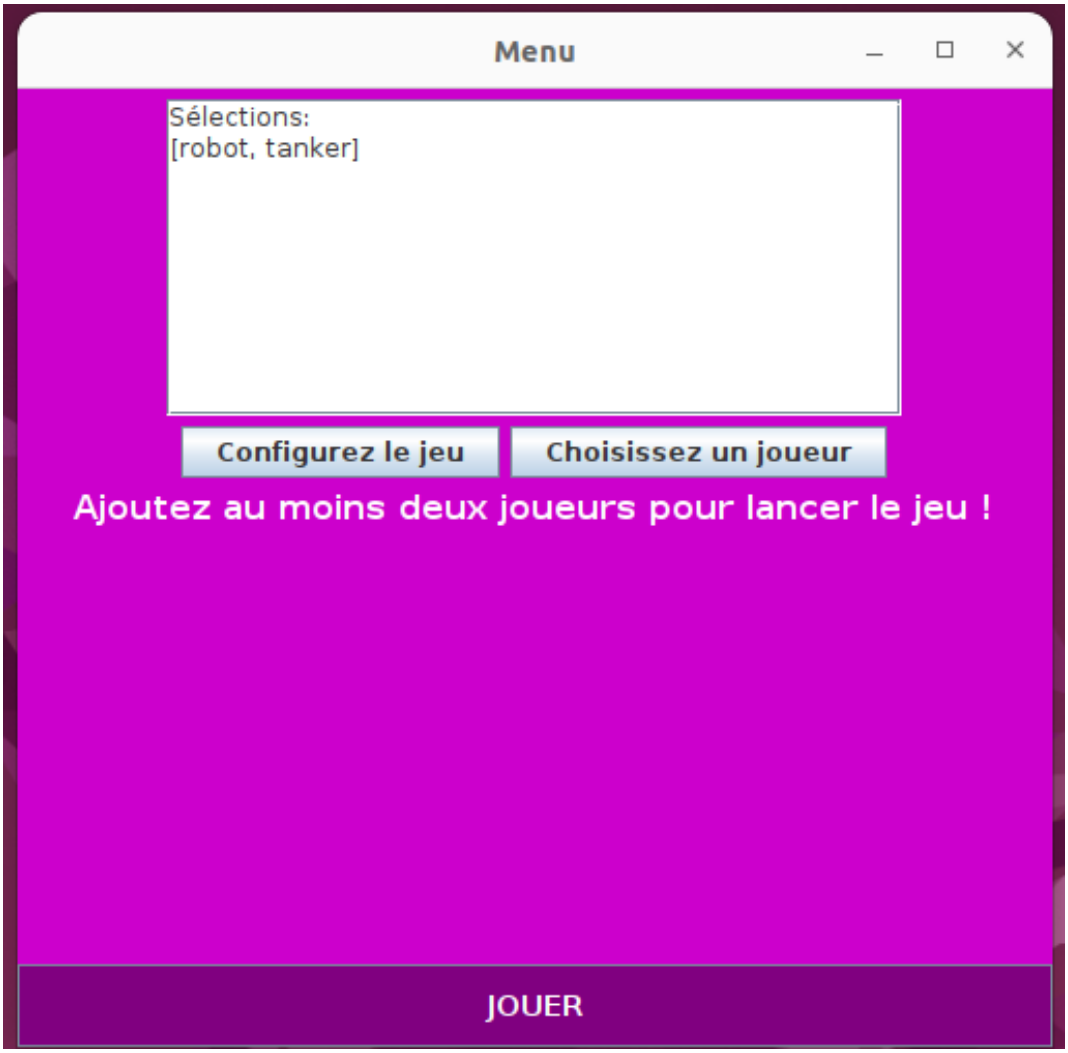

Figure 17 – Menu avec ajout de joueur

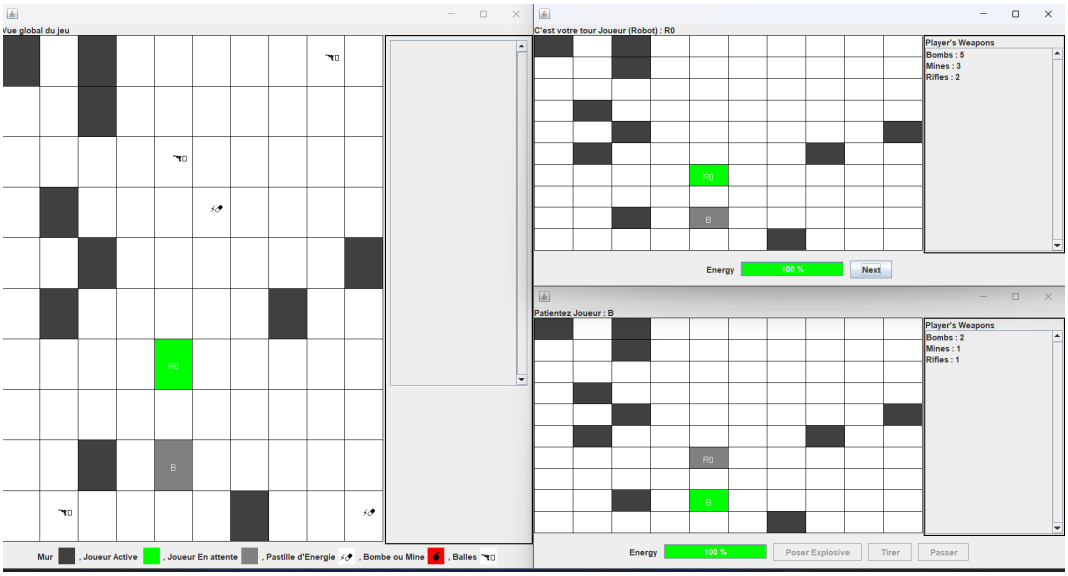

Figure 18 – Vue globale

6 FICHIERS DE CONFIGURATION

Notre application est facilement paramétrable. Nous avons pour cela utiliser des fichiers de configuration .xml. On pourra y définir la taille par défaut de la grille, les paramètres par défaut des joueurs (points de vies, puissance du bouclier...) ou les configurations des différents types de joueur (demolisher, tanker...). Ils sont placés dans les dossiers res/configfiles au même niveau que src.

Pour lire les fichiers, nous avons utiliser des SaxParser. À la classe gameplayers.util.configmanagers.ConfigManager qui s'instancie avec le chemin du fichier de config, on pourra demander une valeur (méthode int getValue (String targetedElement, String targetedParent)) ou des valeurs (méthode List<Integer> getValues (String targetedElement, String targetedParent)) en indiquant la balise cible de l'information et une balise parente de cette dernière.

N.B Le fichier "res/configfiles/default_config.xml" est nécessaire pour les configuration par défaut des joueurs et le fichier "res/configfiles/config1.xml" pour la FighterFactory.

7 TESTS

Des tests ont été effectués pour attester de la robustesse du notre projet. Ils pourront être lancée en exécutant les MainTest dans chaque package de test Cependant, les tests écrits de certains éléments ne sont pas forcément évidents. C'est le cas de la vue, du controller et des stratégies (celles-ci ayant souvent recours au hasard). Pour ce qui est de ces cas, nous nous sommes assurer du bon fonctionnement en exécutant maintes fois nos applications.

8 Conclusion

Au terme de ce projet, nous avons réussi à concevoir et implémenter un jeu de combat tour par tour robuste et extensible. Notre réalisation se distingue par plusieurs aspects clés :

- Une architecture MVC stricte qui sépare clairement le modèle des interfaces utilisateur, permettant d'ajouter facilement de nouvelles interfaces sans modifier le cœur du jeu
- Une utilisation pertinente des design patterns (Factory, Strategy, Proxy) qui rend le code modulaire et facilement extensible
- Un système de configuration flexible via des fichiers XML qui permet de paramétrer facilement le jeu
- Une gestion efficace de la visibilité partielle des éléments du jeu grâce au pattern Proxy
- Des stratégies variées pour les robots, allant du comportement aléatoire à des tactiques plus élaborées
- Deux interfaces utilisateur complètes (CLI et GUI) démontrant la flexibilité de notre architecture