Rapport de projet

Baptiste CLOCHARD

25 novembre 2022

Rapport de développement du jeu de balle au prisonnier.

Développé dans le cadre du cours de conception agile de projet informatique.

Encadrant : Monsieur Valentin Lachand.

Master 1 informatique : année scolaire 2022/2023.

Université Lumière Lyon 2

Table des matières

1 Introduction

1.1 Contexte

Dans le cadre du cours de *Conception agile de projet informatique* j'ai développé un jeu de balle au prisonnier en java en utilisant la librairie graphique JavaFX. Le but du projet est de mettre en pratique les patrons de conceptions et d'architecture vue en cours, d'utiliser git et de programmer une application modulaire.

2 Patrons de conception et d'architecture

Le but des patrons de conception et d'architecture est de répondre à des problèmes d'organisation du code qui arrivent assez régulièrement. J'ai donc pour chaque patron détaillé les raisons de leurs utilisations, c'est à dire quels problèmes ils résolvent.

2.1 Patron d'architecture MVC

Pour rendre le code de l'application plus modulaire, notre code respecte le patron d'architecture MVC (Modèle Vue Controlleur). Le but de ce patron est de dissocier :

le modèle contenant les données et la logique de manipulation de ces données ;

la vue permettant d'afficher les données du modèle ;

le controlleur qui interprète les actions utilisateurs pour modifier le modèle.

Il existe plusieurs manières d'implémenter le patron MVC, j'ai mis en œuvre ici la variante utilisant le patron de conception observer.

Nos interactions entre modèle, vue et contrôleurs peuvent se représenter ainsi :

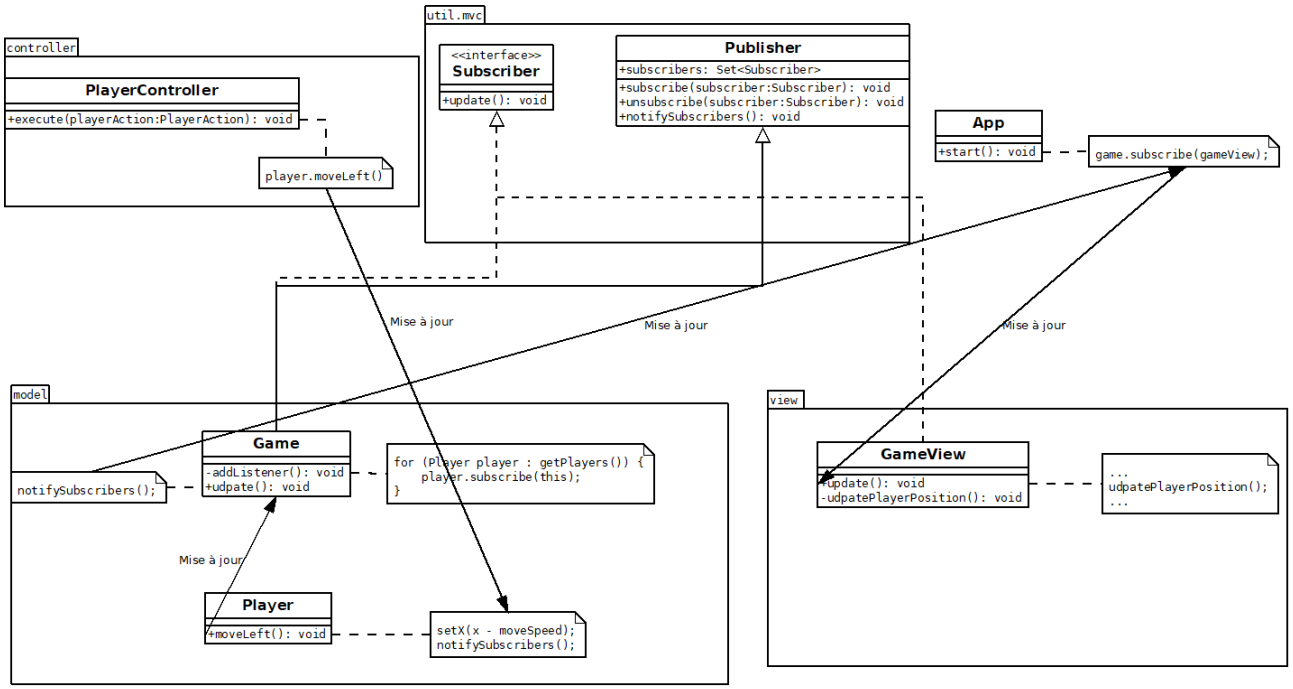

FIGURE 1 – Diagramme UML illustrant le mise en place de MVC en utilisant observer.

Lors de la mise en place de observer deux solutions ont étés envisagées :

- chaque éléments de la vue écoute son équivalent du modèle (cad PlayerSkin écoute Player) ;

- gameVue écoute game.

L'avantage de la première solution est la performance, si seuls les éléments de la vue qui en ont besoins s'actualisent, il y a logiquement de meilleurs performance.

L'avantage de la deuxième solution est qu'il n'y a qu'un lien entre le modèle et la vue. Dans l'optique de séparer le modèle et la vue au maximum j'ai donc choisi la seconde solution. Elle force à avoir deux observers pour mettre en place le MVC : Game écoute les éléments qui composent le modèle (comme Player). Lors d'un changement $game.update()$ est exécutée et toute la vue est alors actualisée.

2.2 Patron de conception de création

2.2.1 Builder

Au fur et à mesure de l'ajout de fonctionnalités dans la classe Player, son constructeur s'est lui aussi agrandi, certains arguments sont facultatifs. La classe Player contenait donc beaucoup de constructeurs :

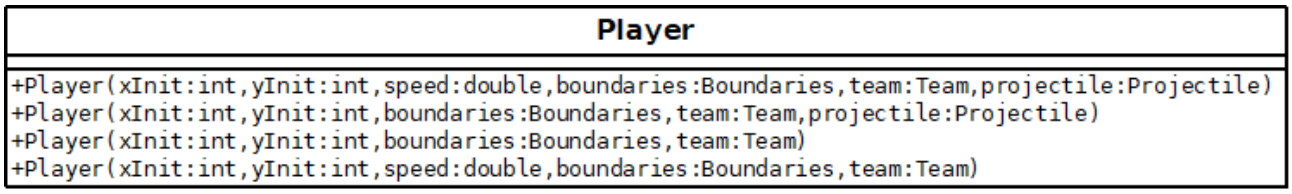

FIGURE 2 – La classe Player contient de nombreux constructeurs à cause des arguments facultatifs.

Le patron de conception builder utilise une manière alternative de créer un objet dans le cas où un constructeur prend beaucoup d'arguments, dont certains facultatifs. L'utilisation du patron builder m'a permis de n'avoir qu'un unique constructeur de la classe Player tout en conservant la possibilité de garder certains arguments facultatifs :

Voici le nouveau code de création d'un joueur utilisant le patern :

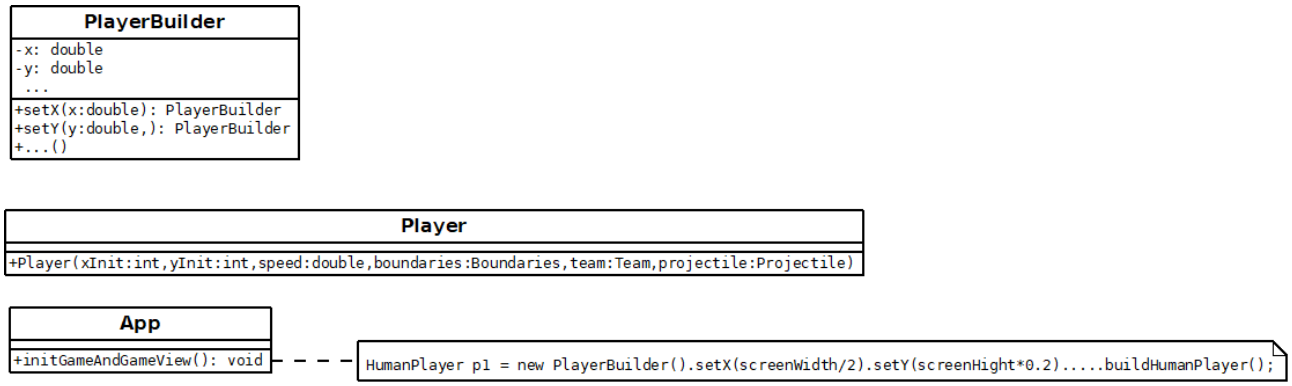

FIGURE 3 – Grâce au patern builder, la création des joueurs est beaucoup plus simple et il n'existe plus qu'un unique constructeur dans la classe Player.

2.3 Patron de conception de structure

2.3.1 Proxy

Le patron de conception proxy permet de contrôler l'accès aux données, il est utilisé en général pour une de ces trois raisons :

- la mise en cache ;
- le contrôle d'accès ;
- l'accès local à un service à distance.

Ici il est utilisé pour la mise en cache. Lors des débuts du jeu, le code générait plusieurs balles et à chaque déplacement de balle l'image était rechargée depuis le disque, ce qui entraînait un ralentissement considérable de l'application.

Notre proxy permet donc de mettre en cache les accès au disque en se basant sur les chemins d'accès pour identifier si un fichier a déjà été chargé ou non.

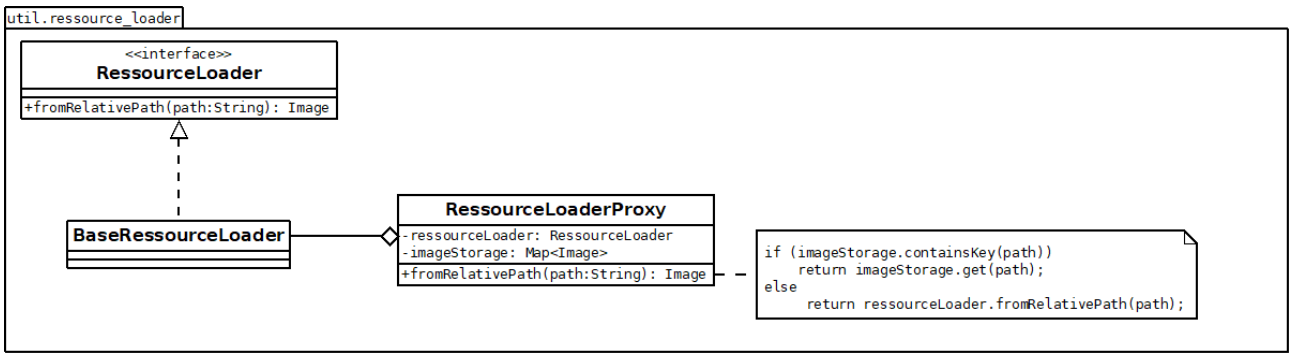

FIGURE 4 – Utilisation d'un proxy pour mettre en cache les accès au disque

2.4 Patron de conception de comportement

2.4.1 Command

Le patron de conception command permet de dissocier les "invokers" (les classent qui vont demander l'exécution de la commande comme un bouton), de la logique de l'application. Le bénéfice est double :

- respecter MVC, ce n'est pas à la vue d'appeler dans le code du bouton directement le code du modèle ;
- éviter le code en double dans notre cas la touche échap et menu > quitter effectuent la même action par exemple.

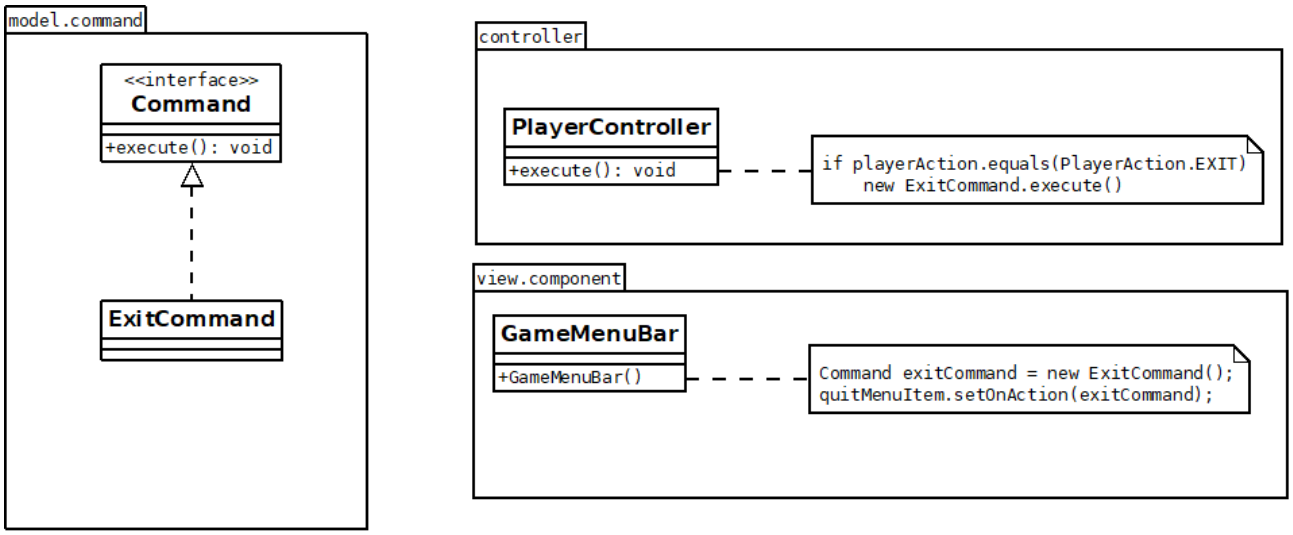

FIGURE 5 – Utilisation du patern command

Le patern command est utilisé dans la figure 5 permet d'éviter de dupliquer du code à deux endroits. De plus, la classe $Command$ extends de $EventHandler < ActionEvent >$ de JavaFX (non visible sur le diagramme) pour pouvoir utiliser $setOnAction(Command)$ directement.

2.4.2 Observer

Le patron de conception observer permet d'effectuer une opération après la mise à jour dans un objet sans avoir à écouter ce dernier en permanence.

Dans le jeu j'ai utilisé la variante de MVC utilisant observer. Le rôle d'observer est d'actualiser la vue lors d'un changement d'état du modèle.

L'utilisation du patron observer est illustrée à la figure 1.

2.4.3 Strategy

Le patron strategy permet de créer plusieurs algorithmes et de choisir lequel utiliser au moment de l'exécution. Dans notre cas, j'ai utilisé ce patron pour pouvoir changer dynamiquement le comportement des IA pendant la partie.

Voici l'ancienne structure de mes classes d'IA :

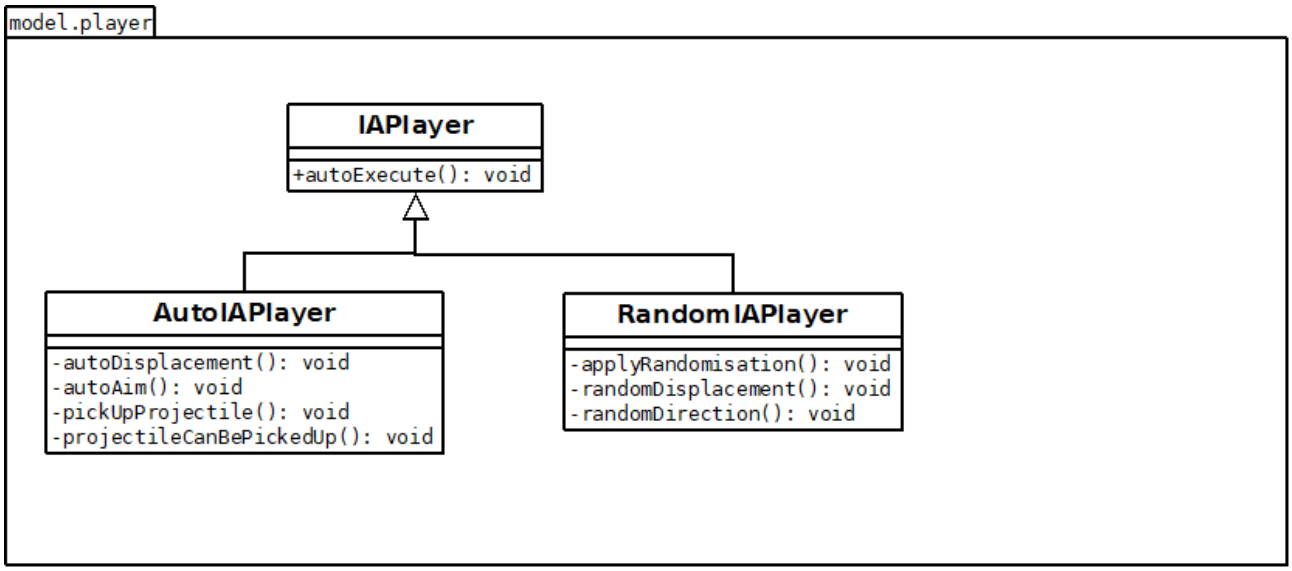

FIGURE 6 – Structure des classes d'IA avant l'utilisation du patron strategy.

Les deux IA alors implémentées étaient dans deux classes différentes et seule l'interface IAPlayer permettait de faire le lien entre les deux. Lorsque j'ai voulu implémenter le changement d'IA en cours de partie, j'ai tout d'abord pensé à remplacer dans la liste des joueurs de la classe game les anciennes IA par les nouvelles. Un problème s'est alors posé : en remplaçant l'objet IA, je perdais la référence dans mes dictionnaires tels que $Map < Player, PlayerSkin > playerSkin$;

J'ai donc réfléchis à une solution pour changer le comportement de l'IA en cours de partie et j'ai utilisé pour cela le patern strategy.

J'ai ensuite découpé les strategy des IA en trois, une stratégie de déplacement, de visée et de tir.

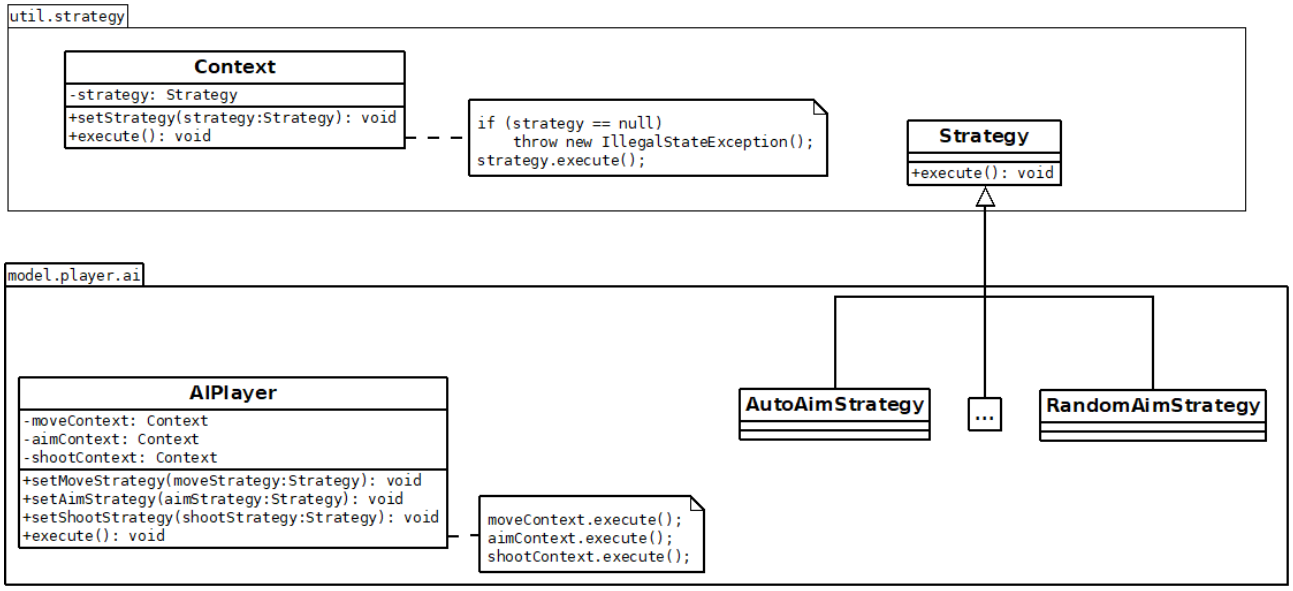

FIGURE 7 – Utilisation du patron strategy pour définir les stratégies d'une IA lors de l'exécution.

On peut désormais changer au moment de l'exécution l'un de ses 3 paramètres pour chaque IA sans effet de bord.

2.4.4 Visitor

Le patern visitor permet lors du parcours d'une structure d'effectuer un traitement en fonction du type de l'objet tout en ayant la logique à l'extérieur des classes de l'objet.

Au moment de coder les collisions, la solution naïve consistait à ajouter les méthodes de collision directement dans les classes Player et Projectile. Cependant ce code n'avait rien à faire dans ces classes, car il était extérieur à leur logique. J'ai donc utilisé le patron visitor pour externaliser le traitement des collisions et obtenir un effet différent en fonction d'une collision avec une balle ou un joueur.

Un second problème s'est alors posé : une collision entre un joueur et un joueur n'a pas les mêmes conséquences que les collisions entre un joueur et une balle. De plus, la conséquence de la collision entre un joueur et une balle dépend de l'état de la balle (en mouvement/statique ...). Pour remédier à ce problème, j'ai ajouté une méthode $getCollisionVisitor(...)$ qui retourne quel visiteur utilisé en fonction de l'objet source.

Un dernier problème c'est alors posé, lors de la collision entre une balle et un joueur, il y a aussi une collision entre un joueur et une balle. En théorie j'aurais dû mettre le code modifiant la balle dans son visiteur et de même pour celui du joueur. Cependant, le joueur doit disparaître *puis* la balle doit s'immobiliser. Dans le cas contraire ça veut dire que le joueur collisionera une balle "immobile" et il va la ramasser au lieu de disparaître. L'obligation de faire le traitement dans cet ordre a rendu impossible la séparation de la logique dans deux visiteurs et tout le traitement a été mis dans un des deux.

Ne voyant pas de meilleure solution j'ai laissé un commentaire dans mon code expliquant pourquoi le code de collision entre balle vers player est vide.

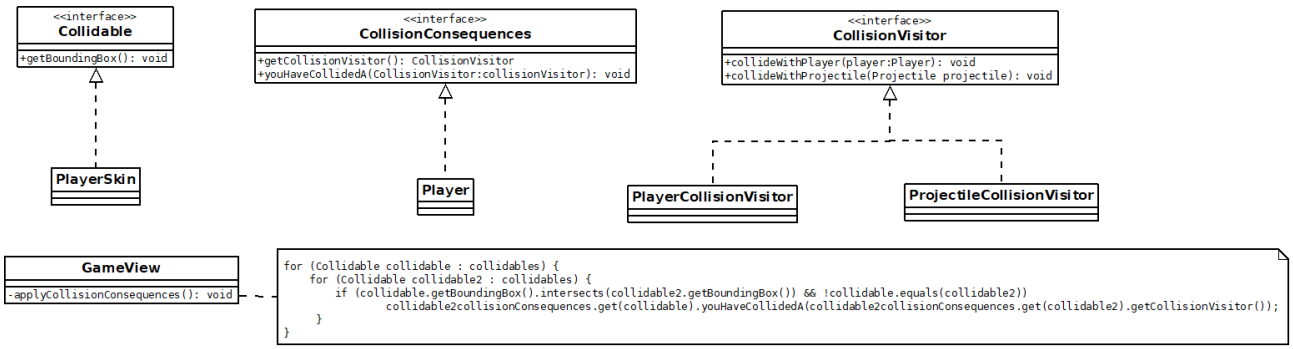

FIGURE 8 – Utilisation du patron visitor pour gérer les collisions

3 Stratégies d'IA

Comme détaillé à la section 2.4.3, nous avons découpé les stratégies des IA en trois stratégies.

Pour l'IA aléatoire :

déplacement aléatoire avec déplacement ou non et changement droite/gauche ;

visée aléatoire avec rotation ou non et changement droite/gauche vers le haut ;

tir aléatoire où l'IA va tirer au bout d'un temps aléatoire.

Pour l'IA automatique :

déplacement automatique avec alternance droite/gauche sur tout le terrain, elle ira chercher la balle directement si elle est immobile de son côté du terrain ;

visée automatique avec tracking d'un des joueurs de l'équipe adverse ;

tir automatique l'IA tire dès qu'elle possède la balle.

4 Améliorations

Voici la liste des améliorations suggérées qui ont été implémentées :

- la mise en pause et le redémarrage de l'application est disponible via Game > Start/Pause ;
- la balle rebondit si elle touche un mur ;
- deux stratégies d'IA qui peuvent être changées dynamiquement via IA > Random/Auto (la partie commence avec une IA de chaque) ;
- ajout de vues supplémentaires avec le score (le nombre de joueurs restant), le nombre d'images par secondes (en haut à droite) et une affichant l'équipe gagnante ;
- la vitesse du jeu est constante peut importe le nombre d'images par secondes, cela évite qu'en cas d'utilisation d'un écran 144hz ou d'une chutte du nombre d'image par seconde la vitesse du jeu soit modifiée.

La suite de cette section porte sur des améliorations qui n'ont pas été suggérées.

4.1 Utilisation de patrons de conception supplémentaires

Un total de 6 patrons de conception et d'un patron d'architecture ont étés utilisés dans le projet. Leur intérêt est détaillé à la section 2.

4.2 Portabilité

Il existe plusieurs façons de lancer le projet.

4.2.1 Lancement par Maven

Il est possible de lancer directement le projet à l'aide de la commande mvn javafx:run exécutée depuis un cmd Windows à la racine du projet.

4.2.2 Exécutable Windows généré avec gluon

il est possible de créer un fichier exécutable pour Windows, pour cela on peut utiliser la commande mvn gluonfx:build. Il est généré dans le dossier balleauprisonnier / target /gluonfx/x86_64−windows et peut être lancé par simple double clic sur le .exe (la présence du dossier assets est nécessaire à côté du .exe).

Un certain nombre de prérequis sont nécessaires pour permettre la génération de l'exécutable :

- l'utilisation de GraalVM ;
- l'installation des outils msbuild et du SDK Windows fournis par Microsoft VisualStudi Installer ;
- le lancement de la commande dans l'invite de commande Windows x64 Native Tools Command Prompt fo

On peut directement depuis le cmd écrire la commande mvn gluonfx:build gluonfx:nativerun pour exécuter le .exe généré.

Pour plus de détails est disponible dans la documentation de GluonFX.

4.3 Plein écran

Le jeu s'exécute par défaut en plein écran sur l'écran principal, la taille de l'écran est récupérée par JavaFX et le terrain est généré directement à partir de celle-ci.

5 Conclusion

Ce travail m'a permis de mettre en pratique certains patrons de conceptions que j'ai appris en dehors du cours, mais que je n'avais pas encore utilisés dans une application. J'ai également essayé d'obtenir un code le plus propre, compréhensible et structuré possible et je suis content du résultat obtenu. Malheureusement j'ai dû changer de groupe de TP et je n'ai pas trouvé un binôme avec qui effectuer le travail, j'aurais préféré pouvoir tester le pair programming et la plupart des améliorations évoquées dans la section suivantes auraient sûrement été terminées.

5.1 Améliorations possibles

Voici la liste des améliorations possibles à l'application ou au processus de développement :

- l'utilisation du TDD et de l'intégration continue ;
- l'ajout d'autres animations de sprite (tir, explosions) ;
- l'ajout de la mise en pause des sprites lors de la mise en pause du jeu ;
- l'ajout du choix de la vitesse du jeu ;
- si on sort du plein écran l'affichage se décale, ce qui casse les collisions ;
- la taille des images est hardcodé, ce serait mieux si elle dépendait de la taille de la fenêtre.