Projet ~ Refactoring et design-patterns

Novembre 2022

Université Lumière Lyon 2

PRÉSENTÉ À

Valentin Lachand

PRÉSENTÉ PAR

Mathis Dousse
Inès Faivre

Présentation globale du projet

Introduction

Pour ce projet en conception agile de projets informatiques et génie logiciel, il nous est demandé d'effectuer une ré-ingénierie d'un code existant en utilisant les patrons de conception vus en cours. Le code existant mis à disposition porte sur le jeu de la balle au prisonnier. Nous allons donc brièvement expliquer les principes et règles de celle-ci.

La balle au prisonnier est un jeu que l'on connait bien souvent de notre enfance. Il consiste à essayer de toucher les joueurs adverses avec le ballon jusqu'à ce que ceux-ci soient tous éliminés (touchés). Pour ce faire, le joueur est touché uniquement si le ballon est ensuite retombé sur le sol, il va dans la « prison » située derrière le camp adverse. Il emporte le ballon avec lui et essaie de le lancer sur l'équipe adverse. S'il arrive à toucher un joueur, il est libéré.

Lorsqu'un joueur arrive à récupérer le ballon, il ne peut plus se déplacer tant qu'il ne l'a pas renvoyé. Il est possible de toucher plusieurs personnes avec le ballon : tous les joueurs touchés avant que le ballon ne retombe au sol sont fait prisonniers.

Dans notre cas, les règles de la balle au prisonnier s'appliquent hormis le système de prison et la répartition. En effet, si un joueur est touché, il reste au sol et ne peut plus jouer. Le ballon revient à son équipe. Si la balle sort des limites du terrain, le ballon revient à l'équipe adverse de celui qui l'a lancé. De plus, les équipes sont réparties en 1 contre 3 (1 humain, 3 ordinateurs).

Quand une équipe entière se retrouve au sol, l'équipe adverse a gagné !

Organisation du rapport

Nous avons décidé de présenter dans un premier temps le travail que nous avons réalisé puis dans un second temps ce que nous aurions aimé réaliser.
En effet, de part nos différentes backgrounds, la partie dédiée à l'implémentation de design pattern et au re-factoring du code selon l'architecture MVC nous semblait très difficile. De plus, nous n'avons jamais utilisé auparavant la librairie javaFX.

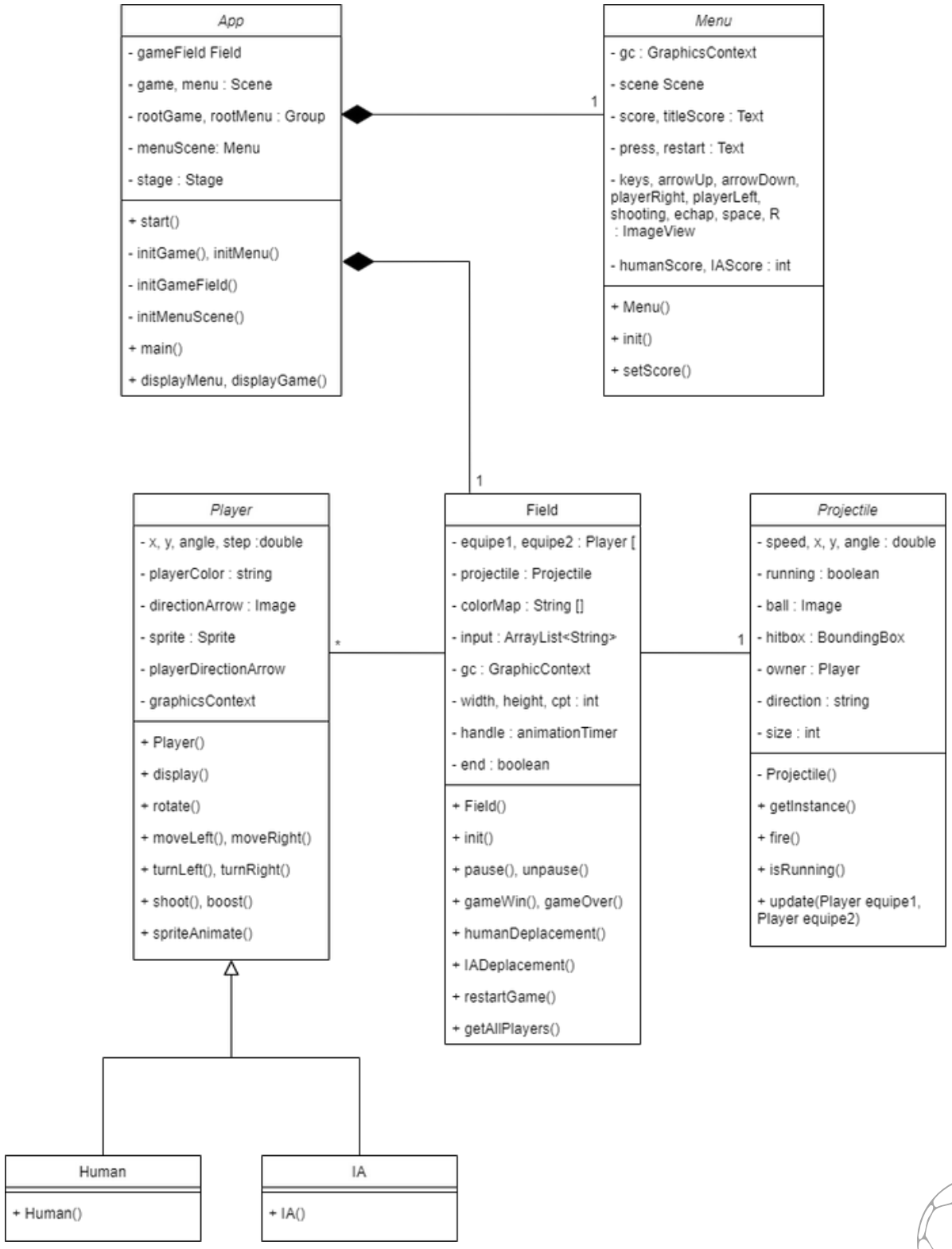
C'est pourquoi nous présenterons l'intérêt d'une architecture MVC dans un tel projet (celui que nous aurions aimé réaliser) et comment il aurait fallu le mettre en place sur le projet de la balle au prisonnier, puis les design patterns qu'il aurait semblé judicieux d'y implémenter. Nous avons vu avec vous pour proposer cette option sur le rapport ainsi que sur le projet. De vous prouver que nous pouvons mettre en place l'architecture MVC ainsi que des design patterns dans un projet Java.

C'est ainsi qu'à côté de cela, nous avons réalisé un projet en JAVA avec la librairie swing mettant en place l'architecture MVC et l'implémentation de plusieurs design patterns. Le principal problème du projet de la balle au prisonnier étant JavaFX, nous souhaitons montrer que nous sommes capables de réaliser une application avec les fonctionnalités du projet initial.
Ainsi, vous trouverez au lien suivant le repository du projet afin de le télécharger : https://github.com/FeckNeck/Pokemon-MVC.
Nous avons aussi réalisé une vidéo présentant le projet ainsi que ses fonctionnalités au lien suivant : https://www.youtube.com/watch?v=FrnxxeMrVHA

Ce rapport comporte une page de plus que demandé dû à la nécessité d'afficher les deux diagrammes assez grands pour leur bonne visibilité.

Organisation du projet

Dans un premier temps, pour la construction de nos différentes classes, nous avons choisi de découper selon le diagramme de classe suivant :
(pour des raisons de lisibilité, les accesseurs et les mutateurs ne sont pas représentés).



Dans ce diagramme, les fonctionnalités de chaque classe sont les suivantes :

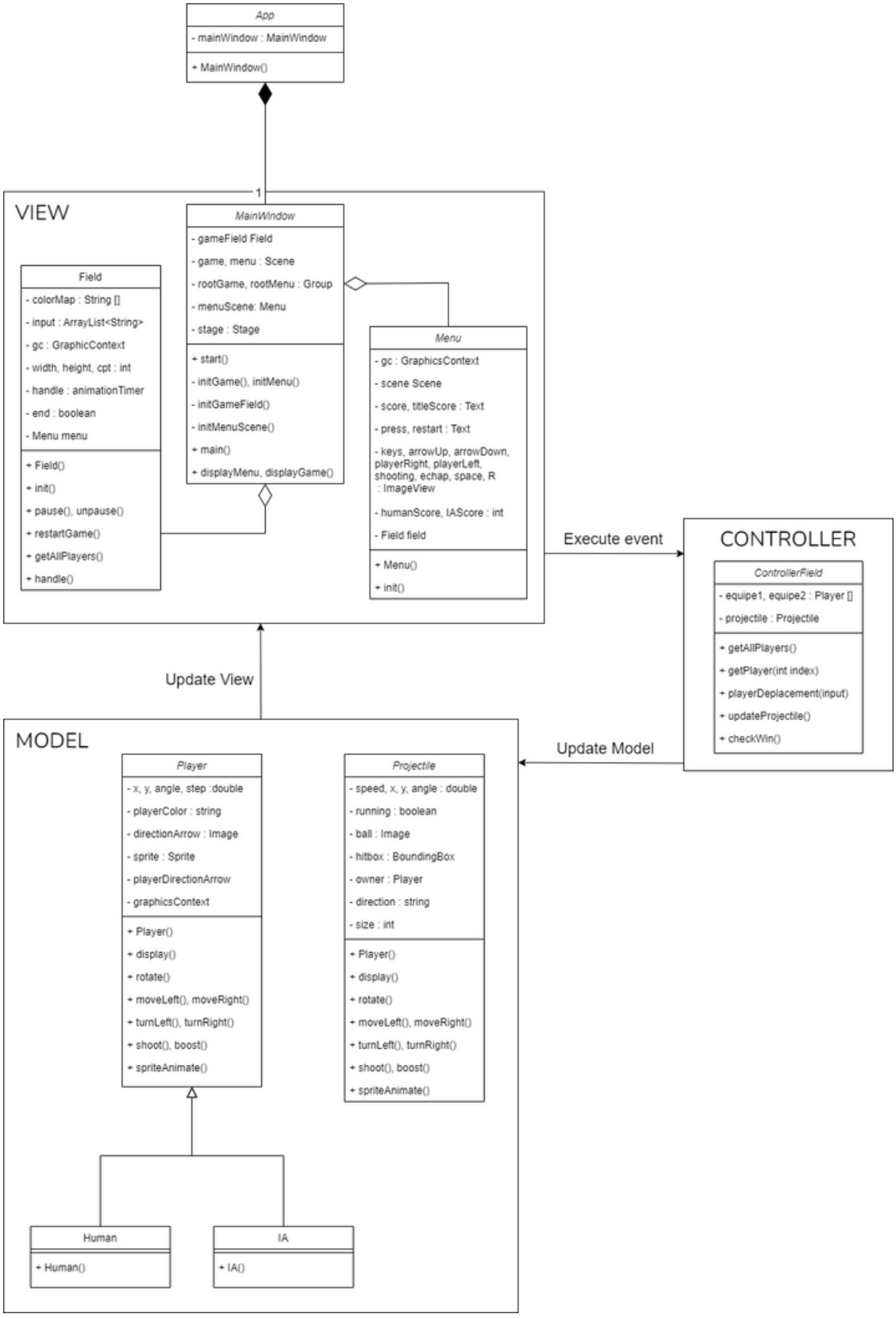
- App est la classe principale du programme, c'est elle qui initialise nos deux scènes : "Field" et "Menu" et qui joue le rôle de médiator entre les deux. Elle permet en effet de faire le lien entre les deux vues afin de passer de l'une à l'autre ou de recommencer une partie depuis le menu.
- Projectile est assez classique, elle permet d'instancier un objet du même nom. Nous avons décidé de lui faire implémenter le singleton car on souhaite initialiser qu'une seule fois un projectile et, dans le cas de la re-création, récupérer celui déjà créé.
- Human et IA héritent de Player afin de pouvoir faire la distinction entre les 2 genres de joueurs.
- Field est la vue qui possède toute la logique de jeu et fait le lien entre le projectile et les joueurs.
- Menu est la vue qui permet d'afficher les commandes de jeu du joueur, recommencer une partie et enfin revenir au jeu.

Concernant les interactions possibles avec l'application :

- Appuyer sur "Echap" permet d'accéder au menu. Depuis le menu, appuyer sur cette même touche permet de revenir au jeu.
- Appuyer sur R depuis le menu permet de recommencer une partie.

Projet abouti

Le diagramme de classe du projet abouti implémentant l'architecture MVC ressemble à ceci :



La principale différence avec le projet abouti est la mise en place de l'architecture MVC :

- Le modèle contient les données manipulées par le programme. Il assure la gestion de ces données et garantit leur intégrité. Ici, les données sont respectivement les joueurs et le projectile.
- La vue permet d'afficher les données récupérées depuis le modèle, c'est elle qui gère les interactions de l'utilisateur. Dans cette application, nous avons 2 vues : celle dédiée au jeu et celle au menu.
- Le contrôler permet la synchronisation entre la vue et le modèle. Il reçoit les évènements de la vue. Si ces derniers demandent une modification des données alors le contrôleur effectue la modification des données du modèle et averti la vue de ces changements.

La mise en place de ce modèle possède de nombreux avantages et est devenue l'architecture par défaut des différents Framework dans tous les langages de programmation : Spring, Angular, Symphony... Tous intègrent l'architecture MVC. Cette dernière permet une meilleure organisation du code où chaque classe a un rôle unique dans l'application. Le code est plus facilement réutilisable par d'autres applications et la possibilité de réaliser des tests unitaires est plus facile et cohérente.

Dans notre application la mise en place du MVC a engendré les modifications suivantes:

- La méthode "handle" de la classe Field ne modifie plus directement les données du modèle mais appelle les fonctions du contrôleur: "playerDeplacement", "updateProjectile" et "checkWin" qui effectuent les tests et modifient ensuite les données.
- L'app initialise simplement notre mainWindow qui initialise notre vue Menu et Field et joue le rôle de médiator entre les deux.
- Enfin, le contrôleur modifie les données du modèle en fonction des interactions de l'utilisateur et met à jour le score selon les conditions. Le menu n'a plus qu'à initialiser le contrôleur et récupérer le score.

Dans cette conception, les design patterns suivants auraient été judicieux à mettre en place :

- **Singleton** : Tout comme la première version du projet, le singleton peut être implémenté pour la classe projectile mais doit surtout être implémenté pour le contrôleur. Le contrôleur va être initialisé au moins 2 fois, dans chacune des 2 vues et il ne faut surtout pas que ce dernier soit de nouveau initialisé car on se retrouverait avec 2 instances de contrôleurs possédant chacun leurs variables. Lorsqu'une partie va se terminer et que le score va changer dans le contrôleur, il faut pouvoir accéder à ce même score depuis le menu. Une nouvelle initialisation empêcherait cela alors que récupérer l'instance résout ce problème.

- **Observer** permet de rafraîchir la vue lorsque les données sont mises à jour grâce à un mécanisme d'abonnement. Ainsi, lorsqu'un joueur se fera toucher, l'observer se chargera de cacher ce dernier dans la vue.

- Le **prototype** aurait pu être implémenté sur nos joueurs afin de rendre leur initialisation dans la class Field plus propre. Il aurait été alors possible de cloner les joueurs "humain" et "IA" plutôt que d'en faire des nouveaux.

- Afin de gérer nos joueurs, il aurait été possible de créer une **factory** qui aurait pu permettre de gérer le type de joueurs à instancier en fonction du besoin. C'est une approche différente du prototype mais qui aurait pu être une bonne idée dans le cas d'un jeu avec plus de joueurs.

- Le **médiator** était déjà implémenté au travers de l'app dans la première version et ici à travers la mainWindow. Il permet de faire le lien entre 2 classes qui doivent échanger entre elles. Ici, nos 2 vues doivent interagir entre elles notamment lorsque l'utilisateur veut accéder au menu depuis le jeu et inversement. Le médiator permet de résoudre le problème en intégrant les instances des 2 vues.