Rapport

Conception agile de projets informatiques et génie logiciel

Refactoring et design-patterns

Hugo BENSBIA, Moïse BERTHE

29/11/2022

Table des matières

I. Présentation

Les design pattern sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Ce sont des sortes de plans ou de schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code. Le projet consiste dans un premier temps d'effectuer, à l'aide de ces solutions, une réingénierie du code d'un jeu de balle au prisonnier afin de mieux structurer le projet et le rendre plus modulaire en séparant la partie logique métier et les couches graphique et en rendant les différentes classes indépendantes les unes des autres.

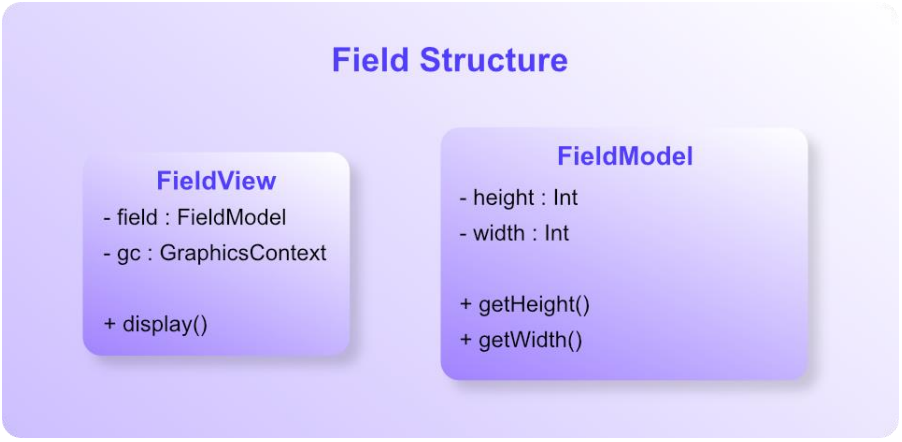
Dans un second temps d'étendre les fonctionnalités du jeu à l'aide : d'un mécanisme de contrôle de collision entre les différents composants graphiques, de joueurs manipulés par une intelligence artificielle et de vues permettant de suivre et contrôler le jeu.

Nous avons principalement travaillé sur la première partie qui concerne la réingénierie du code en adoptant les design pattern MVC (Model View Controller) et singleton.

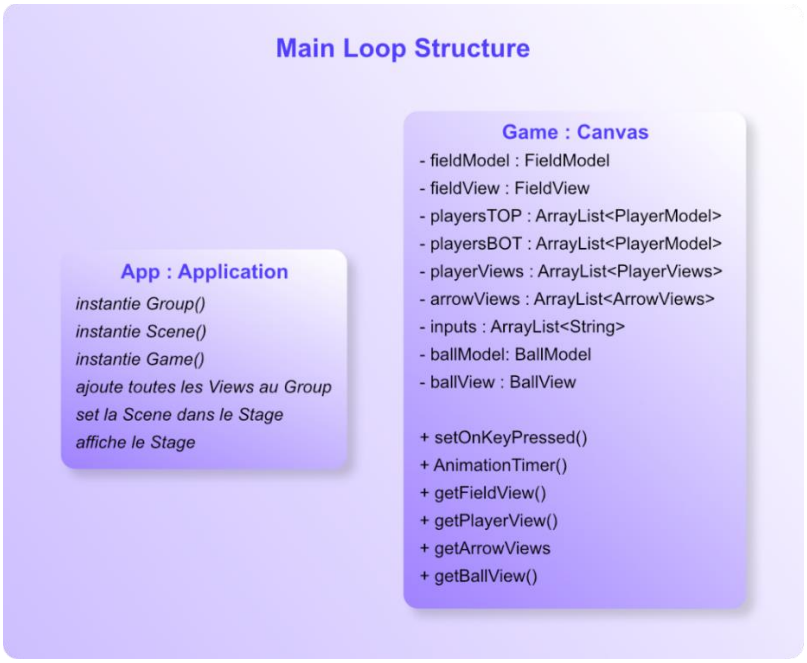
II. Architecture

1. MVC

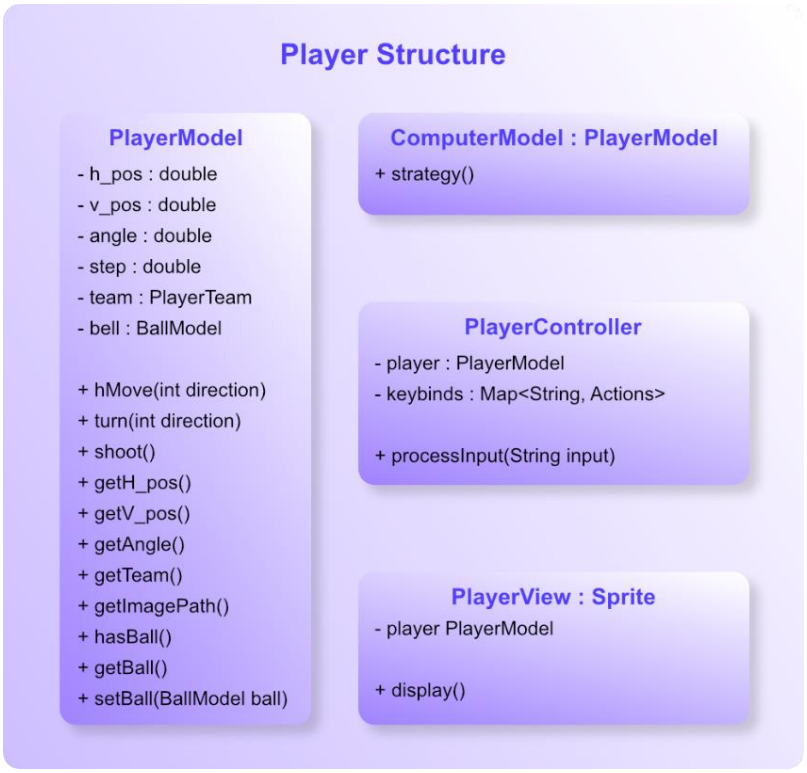
Nous avons adopté l'architecture MVC car il fournit un découpage parfait du projet entre la partie logique métier et les couches graphiques. Les éléments du jeu sont donc mieux organisés avec notamment la séparation de l'affichage des joueurs et la gestion de leurs mouvements.



Nous avons une classe **Game** qui hérite de **Canvas** dans lequel on a accès à tous les composants graphiques. L'écouteur d'événement est posé sur cette classe ce qui nous évite de le poser uniquement sur un de nos composants graphiques par exemple sur **Field**.

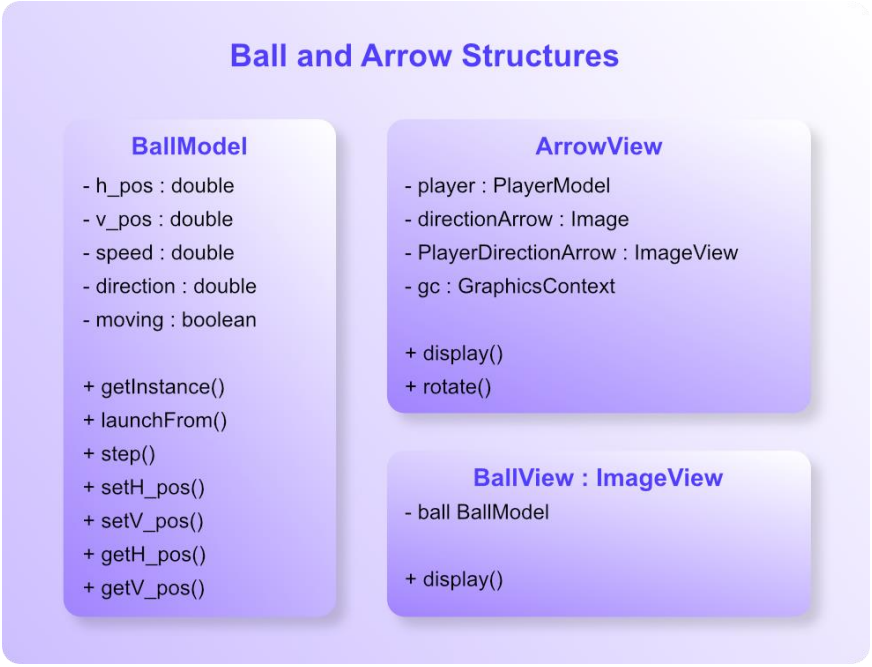


La classe **PlayerView** hérite de **Sprite** pour pouvoir être directement ajouter à la liste des éléments qui sont rendus dans notre scène. Ce choix nous est paru plus pertinent que de donner un attribut de type **Sprite** à **PlayerView**.

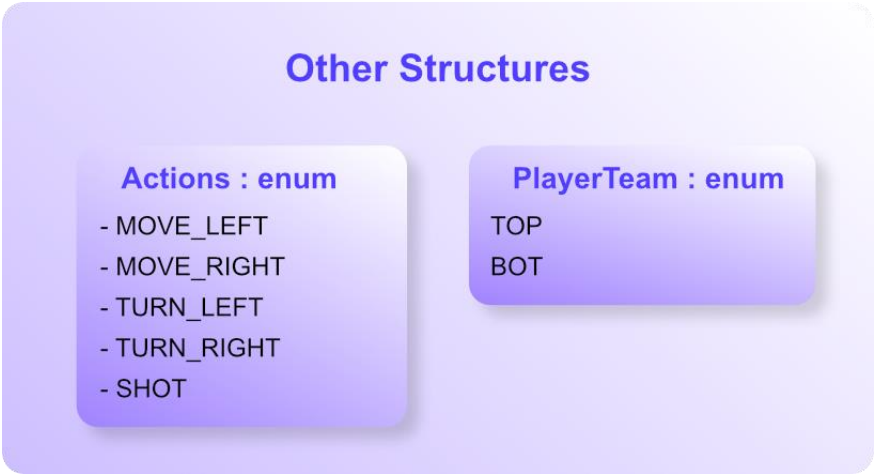


La classe **BallView** hérite de **ImageView** pour la même raison que **PlayerView** hérite de **Sprite** ça nous permet de directement ajouter un objet **BallView** à notre scène.

La classe **ArrowView** est utilisée pour effectuer des transformations (rotation) sur l'image de la flèche et de faire le rendu dans notre scène.



Les énumérations **Actions** et PlayerTeam permettent de définir un ensemble fini de constantes qui décrivent respectivement les actions que peuvent effectuer un joueur et l'équipe auquel le joueur appartient.



2. Singleton

Ayant qu'une seule balle dans le jeu, le design pattern singleton est utilisé pour gérer cette balle. Nous permettant ainsi de n'avoir qu'une seule instance de la balle tout au long d'une partie de jeu.

III. Ressentis

Personnellement, j'ai trouvé le projet intéressant du point de vue architectural, mais maladroit du point de vue du langage. Ma plus grande difficulté a été de me plonger dans le paradigme de Java avec les classes omniprésentes, ce qui a grandement ralenti notre re-factoring. Finalement, cela ne m'a pas empêché d'apprendre de nouveaux outils de programmation (les design patterns). Dorénavant, durant la phase préparatoire d'un projet, je me sens capable de cerner ces patterns pour pouvoir les implémenter de manière conventionnelle. **[Hugo Bensbia]**

N'ayant aucune expérience en développement d'interface graphique avec Java prendre en main JavaFx était assez compliqué. Néanmoins j'ai trouvé le projet intéressant surtout la mise en œuvre des design pattern et plus précisément l'utilisation du MVC en développement autre que le développement web. Je me sens capable d'identifier une situation nécessitant l'utilisation d'un design pattern et d'utiliser la solution adéquate. **[Moïse Berthé]**

IV. Conclusion

C'est un projet où nous avons eu les yeux plus gros que le ventre. En effet, repartir de zéro au niveau du code nous a fait passer beaucoup de temps sur la compréhension de Java ou particulièrement JavaFX. Néanmoins, nous sommes plutôt contents d'être arrivé à un résultat fonctionnel en écrivant 100% du code principal (sauf la classe Sprite).

On termine sur un code qui est moins complet que l'on ne l'aurait voulu, mais où chaque élément n'est pas mis au hasard et chaque concept est bien assimilé.