RAPPORT DU PROJET

BALLE AU PRISONNIER

NGUYEN Xuan Huy p1419771

HUYNH Cong Lap p1419778

1. Présentation globale du projet

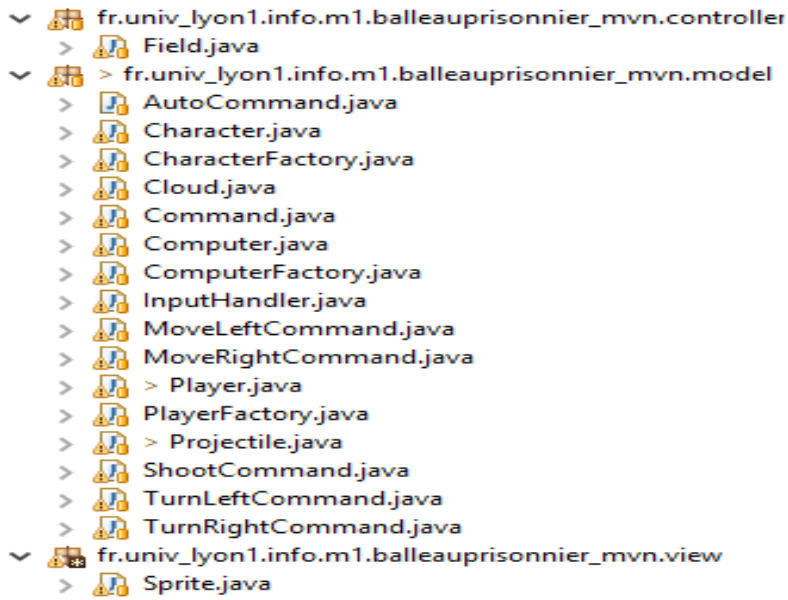
Le projet Balleauprisonnier est un jeu vidéo simple développé par le language Java. Avec les stratégies simples, on a deux équipes possèdant 3 personnages (1 Player et 2 Computer) et les obstacles (les nuages). En fait, n'importe type de personnage peut tirer les balles pour détruire ses concurrents. Pourtant, les obstacles pourraient changer la direction de balle tiré par le personnage.

2. Architecture du projet

Ce projet est construit sur la base du modèle MVC. En plus, les deux patterns Abstract Factory et Command sont appliqués également dans ce projet.

2.1 MVC

Le développement du jeu et de l'UI comporte le flux de travail habituel d'attente pour la saisie d'un utilisateur ou d'autres conditions de déclenchement, en envoyant la notification de ces événements à quelque part approprié, en décidant quoi faire en réponse et en mettant à jour les données. Ces actions montrent clairement la compatibilité avec MVC.



Modéle MVC appliqué dans le projet

Controller:

- la classe Field renverrait les entrées d'utilisateur comme pression de bouton, événements de clavier, clics de souris, etc. Puis, elle déciderait quel est le Model va traiter ces entrées. Et après ces traitements, les Vues correspondants seraient appelés.

Model:

- les classes gèrant la manipulation des données (MoveLeftCommand, Character, etc) mettraient en place dans la partie Model. Ces classes traiteraient les problèmes de logique du jeu.

Vue:

- les classes concernant aux animations d'objet comme le personnage, l'explosion resteraient à l'intérieur de la partie Vue. L'animation du personnage marchant, du personnage mort, de l'explosion seraient traités par ces classes.

L'intérêt de MVC: les gestion (déplacements, stratégies, etc), l'animation des personnages, le contrôleur se séparaient mutuellement. Grâce à cela, lorsqu'on ajoute un fichier, par exemple: une fichier concernant au traitement logique, ce fichier serait mis en place dans la partie Model.

2.2 Abstract Factory

Les personnages font une partie intégrante des jeux. La gestion de l'instanciation des personnages est très important parce que grâce au mieux modèle de l'instanciation du personnage, l'expansion du jeu, lorsqu'on ajoute les nouveaux personnages, assurerait le code d'instanciation ne changerait pas trop. Donc, le pattron de création comme Abstract Factory est le plus approprié.

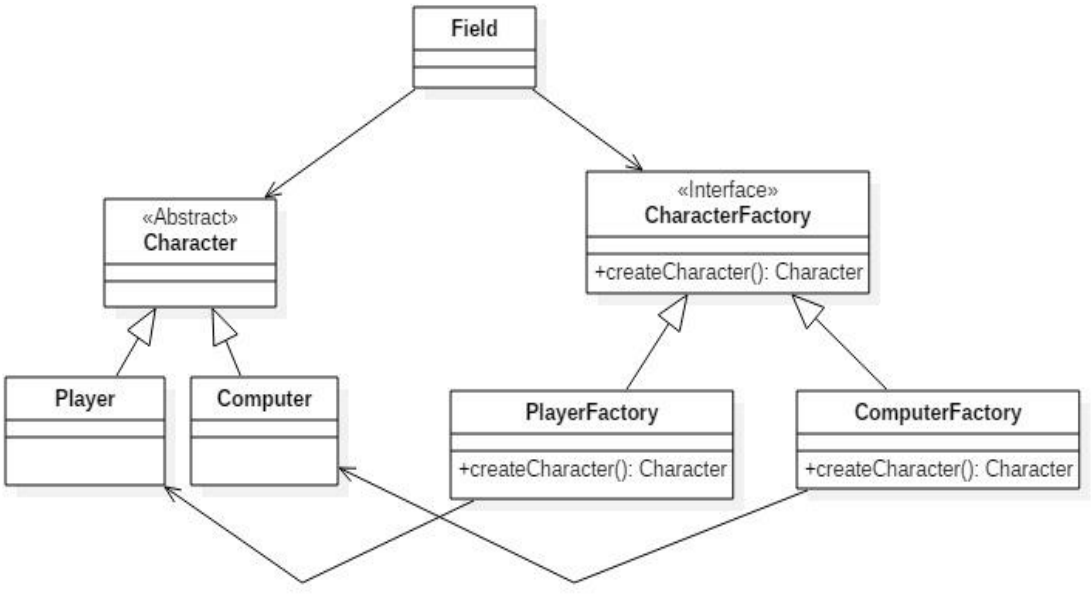


Diagramme de classe: Abstract Factory

La classe Player et Computer hérite la class abstrait Character pour avoir les comportements communs aussi que possèder les comportements propres.

La classe PlayerFactory et ComputerFactory créeraient le Player et le Computer, respectivement.

Grâce à l'interface CharacterFactory, on pourrait créer le Character (Player ou Computer) comme on veut, via le méthode createCharacter(). Et, l'initialisation du Player ou Computer serait masqué.

L'intérêt de l'application du design pattern Abstract Factory: le CharacterFactory nous permet de déporter le choix de la classe concrète à instancier dans un unique objet. Si on a plusieurs types de Character, le code d'instanciation n'oblige pas d'être réécrire à chaque fois.

2.3 Command

Normalement, quelque part dans chaque jeu vidéo est un morceau de code qui se lit l'entrée d'utilisateur – pression de bouton, événements de clavier, clics de souris, etc. Il prend chaque entrée et le traduit à une action significative dans le jeu. De plus, nombreux jeux permettent à l'utilisateur de configurer la façon dont leurs boutons sont mappés. Donc, un objet représentant un action de jeu est nécessaire et le pattern Command est le plus approprié pour ce but.

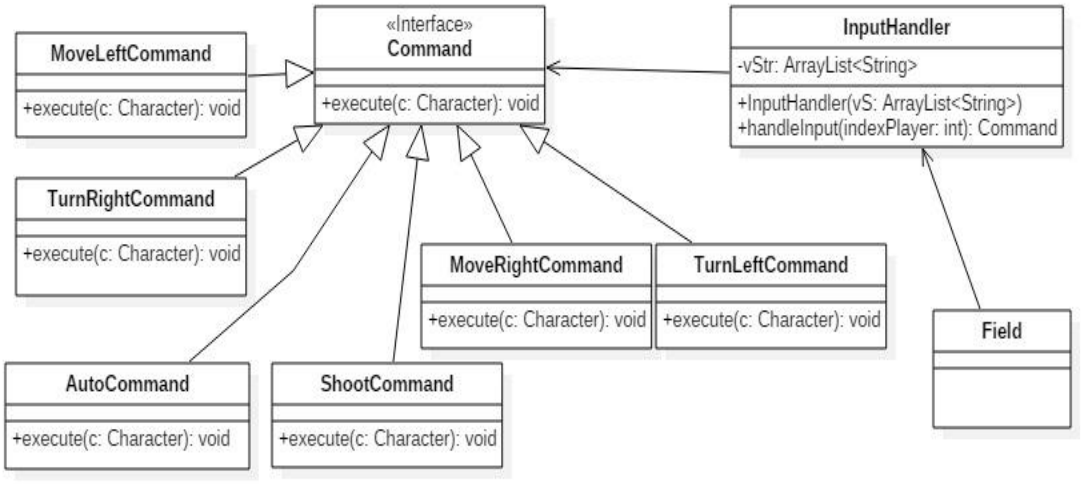


Diagramme de classe: Command Pattern

Les classes MoveLeftCommand, MoveRightCommand, TurnLeftCommand, TurnRightCommand, ShootCommand, AutoCommand va appeler les méthodes moveLeft(), moveRight(), turnLeft(), turnRight(), shoot(), autoMoveTurnShoot() du personnage pour se déplacer à gauche, à droit, tourner la flèche à gauche, à droit, tirer, se déplacer aléatoire, respectivement.

Ici, c (type: Character) est un objet représentant un personnage dans le monde du jeu. Ce objet est passé pour execute(c) afin que la commande dérivée puisse invoquer des méthodes sur un acteur de notre choix.

Grâce à la liste d'input vStr, initialisé par la constructeur par le ArrayList<String>, et l'indice du joueur, le méthode handleInput() pourrait invoquer des commandes appropriés.

Comme la valeur retournée du méthode handleInput est un objet Command, quand sa méthode execute(c) est appelée, elle exécuterait l'action de jeu.

L'intérêt de l'application du design pattern Command: lorsqu'on veut changer le traitement d'une commande, on doit changer seulement la classe de la commande dérivée correspondant sans la modification de l'exécution de la commande. Et si on ajoute les autres commandes, leurs classes hériteraient seulement l'interface Command.