Mathis Gourret
Leo Gondouin

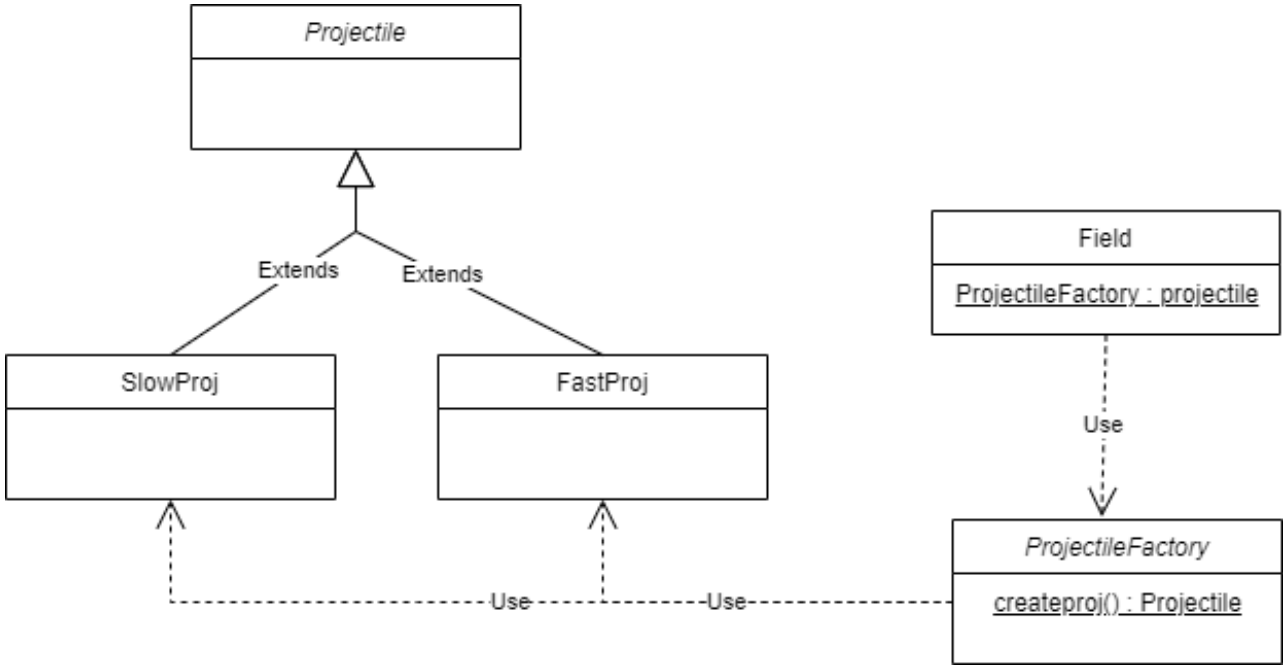
Conception Agile de Projet

Partie 1 : Introduction de notre projet :

Nous avons fait un jeu de la balle au prisonnier qui se joue à 2. Le jeu est décomposé en 2 équipes de 3, une en bas et une en haut. L'objectif est d'éliminer tous les joueurs de l'équipe adverse.
Les équipes sont composées de 1 joueur et 2 IA. Les IA ont 2 modes, l'un est le mode aléatoire où les IA font des actions aléatoires et l'autre où elles suivent les mouvements du joueur.
Pour éliminer les joueurs de l'équipe adverse il faut lancer une des deux balles sur les joueurs de l'équipe adverse. Les balles sont décomposées en 2, une lente et une rapide. Quand une balle ne touche pas un joueur, celle-ci réapparaît dans les mains d'un des joueurs de l'équipe visée de façon aléatoire.
Notre jeu dispose d'un bouton afin de commencer une partie ainsi que d'un bouton pause.

Le projet est divisé en plusieurs classes dont la main est App, mais la majorité des améliorations sont dans la classe Field. Pour la réalisation de ce projet nous avons utilisé les design patterns : Factory et Singleton. Nous n'avons pas réussi à implémenter la méthode MVC par faute de temps et de compatibilité, nous n'avons donc pas l'architecture que nous avions prévue.

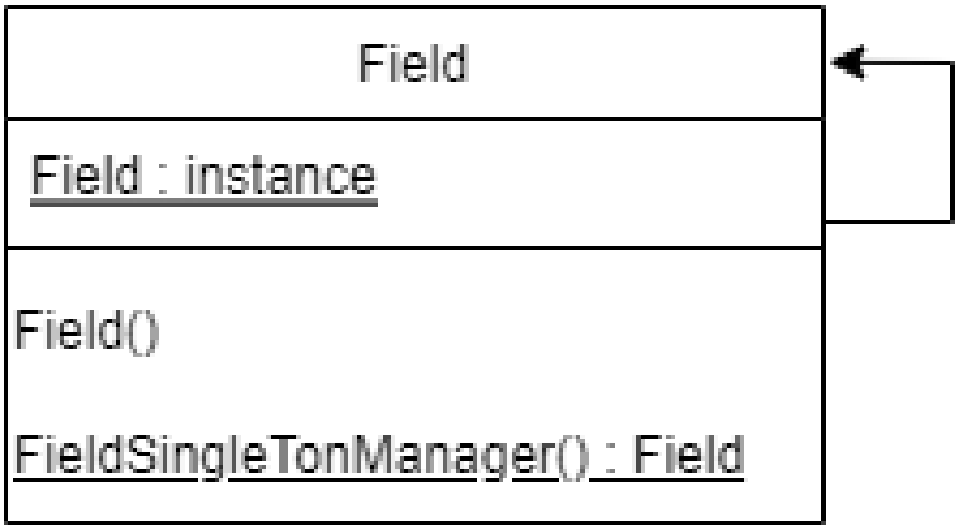
Partie 2 : Le Pattern Factory :



Le design pattern Factory nous permet de ne faire les appels de classes à l'intérieur de la classe Factory plutôt que dans notre programme principal et ainsi nous permettre de créer des "produits" à partir d'une classe originale. Nous permettant de créer des balles avec des effets différents sans faire des milliers d'appels de classes dans notre programme principale.

Nous avons décidé d'utiliser le pattern Factory, avec comme projet de faire plein de balles différentes et qui, à la base, aurait peut-être pu être sélectionné au début. Mais aussi faire que l'on puisse choisir un "lancé" avant de tirer, afin de varier le gameplay.

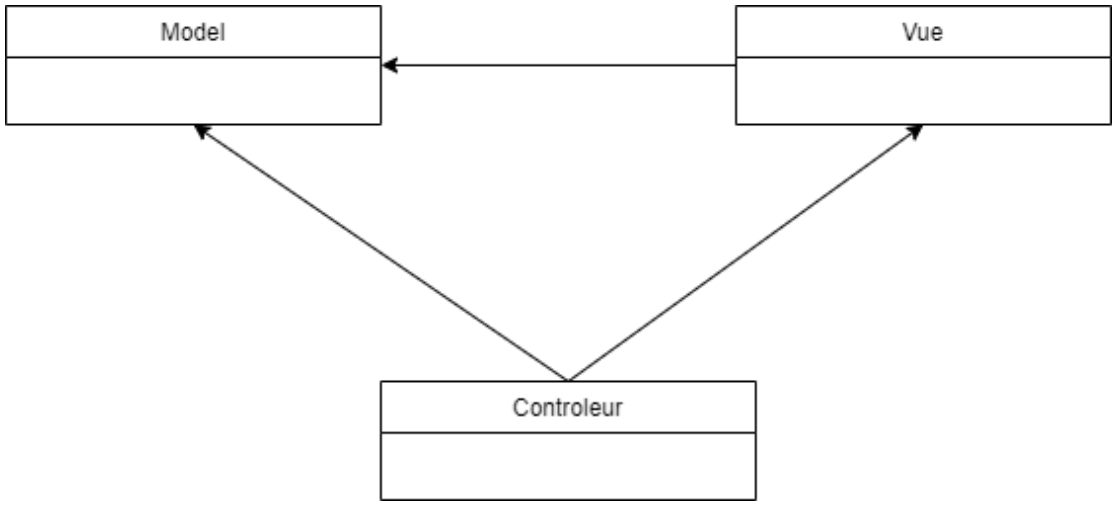
Partie 3 : Le pattern Singleton :



Le pattern Singleton nous permet de rendre l'instanciation d'une classe unique ainsi éviter la multiplication d'objet de cette classe. Et ainsi avoir un accès global à celui-ci.

Nous avons décidé d'utiliser le pattern Singleton afin de rendre notre Field, qui est à la base du fonctionnement de notre projet, unique. Et ainsi qu'il ne puisse pas y avoir de collision ou de gène si une autre instance était créée. C'est aussi pour éviter tout ralentissement si jamais d'autres instances venaient à être créées. Et enfin parce que notre Field doit être le seul Field existant.

Partie 4 : Le modèle MVC :



Le modèle MVC permet de rendre le Modèle (le fonctionnement global du projet) de la Vue(l'interface) à l'aide d'un Contrôleur (la tour de contrôle qui gère tous les changements et les échanges d'informations). Ainsi la Vue utilise le Model pour se créer et le Contrôleur informe le Model que la Vue a changé et aussi si l'inverse se produit. Il permet donc de compartimenter les différentes parties et ainsi réduire les dépendances.

Au niveau de la structure MVC dans l'idéal on voulait faire un Modèle joueur, bots, projectiles et un Modèle Field regroupant l'ensemble des objets dans le même "monde" puis faire une Vue affichant le Modèle Field. Avec GameEngine qui est le Contrôleur gérant les actions et l'affichage de la Vue et les Modèles. Et ainsi avoir un projet plus structuré, plus modifiable et étant moins sujet à la casse.

Partie 5 : Conclusion :

En conclusion, il fallait se limiter aux attentes du client, le fait de vouloir gérer deux types de projectiles différents en même temps ont amenés beaucoup de bugs qui n'ont pas pu être fixés avant le deadline.

Mais ce qui nous gêne le plus est l'implémentation de la méthode MVC, qui nous a fait perdre beaucoup de temps et ainsi nous a empêché de faire tout ce que l'on voulait faire, nous obligeant à nous rabattre sur une architecture linéaire simple en grande partie.

L'architecture MVC aurait permis d'épurer notre code, et de résoudre les bugs plus facilement et rapidement.