Mini-projet SimuLife
Automne 2017-2018

WaterWorld

Groupe No : 6

Fusionné avec : ChessLife

Rendu Final

Fuchs Nicolas
Michel Guillaume
Monney Bastien

Date du rendu 11/01/2018

Enseignant : Pierre Kuonen / Julien Tscherrig

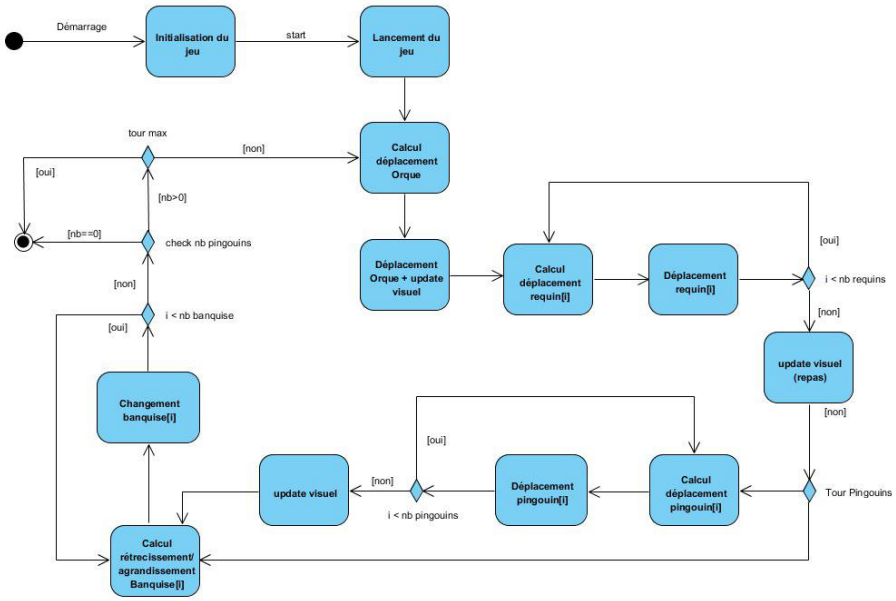
Table des matières

1 Spécifications complètes du jeu 3
2 Diagramme de classe de notre jeu 4
3 Utilisation des patterns 5
3.1 Command 5
3.2 Singleton 5
3.3 Observable 5
3.4 Factory 5
4 Diagramme de classe final après fusion 6
5 Modification diagramme de classe pour fusion 7
6 Diagramme de séquence après implémentation 8
7 Diagramme de classe 9
8 Comparaisons/commentaires 10
9 Analyse personnelle de l'utilisation des patterns 10
9.1 Command 10
9.2 Singleton 10
9.3 Observable 10
9.4 AbstarctFactory 10
9.5 Factory 10
10 Pattern Builder et bridge 11

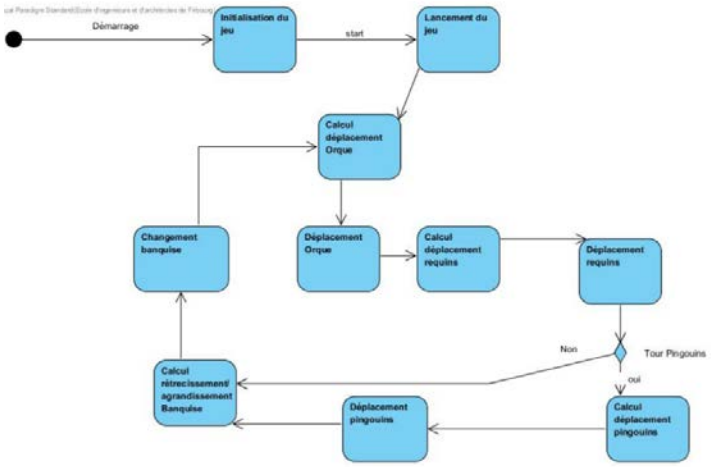
1 Spécifications complètes du jeu

Au niveau du principe de base, des créatures utilisées, des principes du jeu, rien n'a changé. C'est au niveau de notre implémentation qu'il y a eu des changements. On remarque un gros changement entre le fait qu'avant, on avait toutes nos créatures qui dépendaient de l'interface ICreature et qui devaient donc implémenter chacune des variables. Actuellement ce n'est plus le cas, c'est la classe créature qui contient tout cela et les créatures dépendent de cette dernière. De plus, les mouvements ont été changés, auparavant, on avait une mauvaise utilisation du pattern commande avec des noms de classes erronés ou du moins pas optimaux (tel que CommandCaller). Alors que maintenant, l'utilisation du pattern est correcte avec un move spécifique à chaque créature qui dépendent de la classe principale Move.
Pour ce qui est de notre boucle principale, elle a été précisée suite aux remarques de notre professeur. Pour cela, nous avons déroulé plus complètement la boucle pour qu'elle soit mieux comprise, surtout au niveau du mouvement des requins et de la banquise. Nous avons ajouté aussi la fin du programme car notre schéma de base ne représentait effectivement pas de fin à notre jeu.

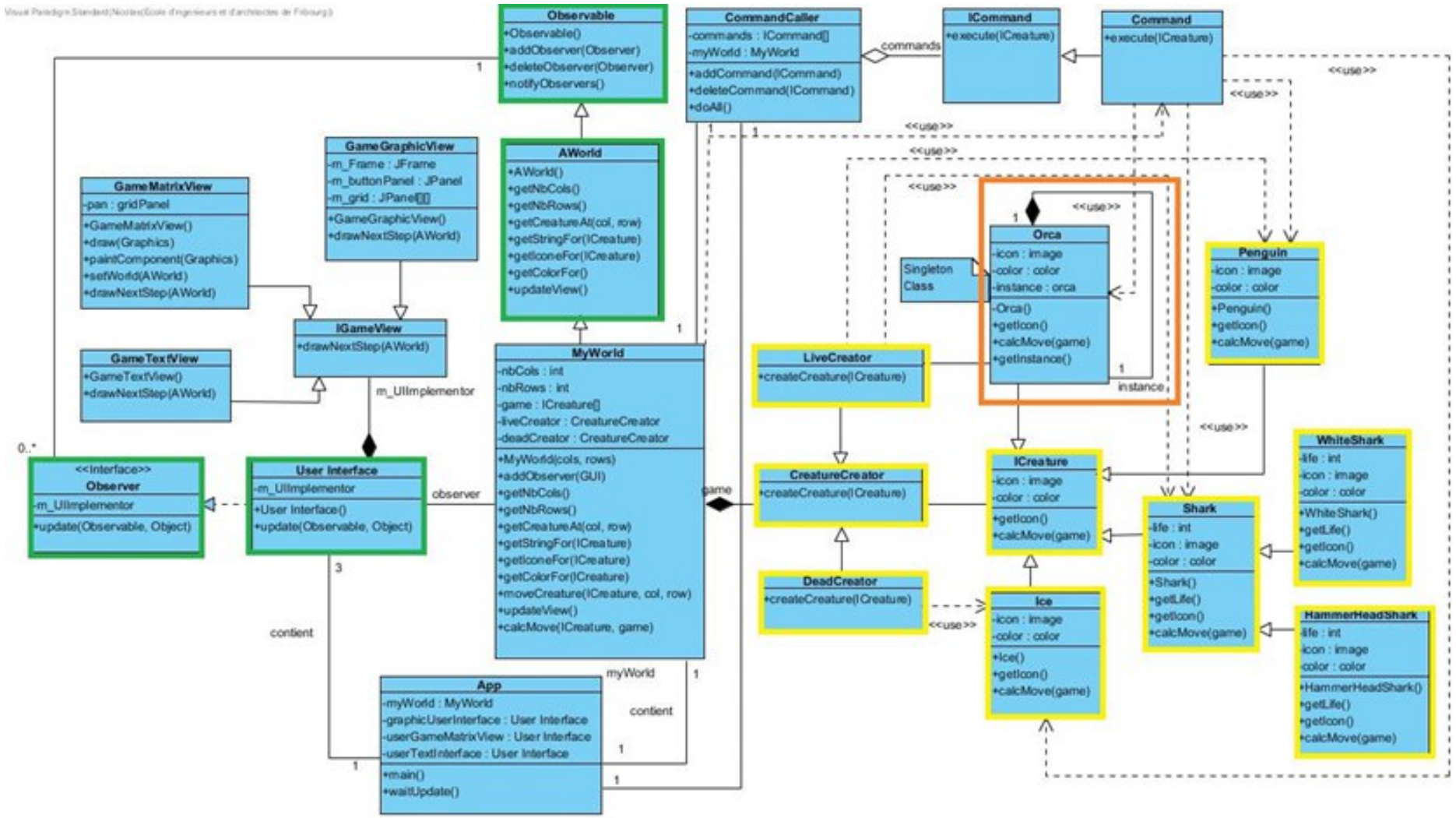
APRES



AVANT

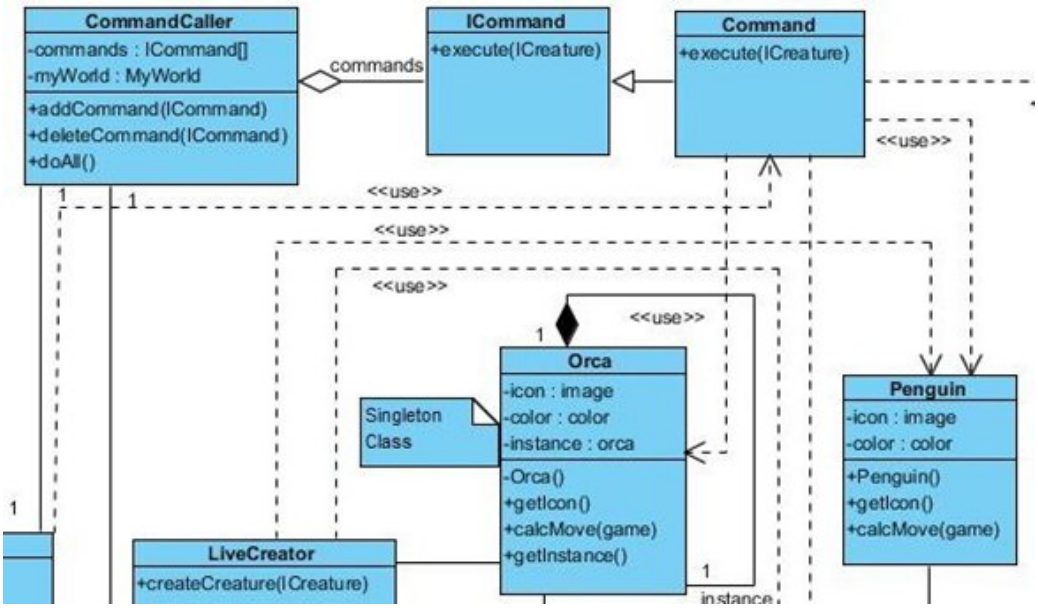


2 Diagramme de classe de notre jeu



3 Utilisation des patterns

3.1 Command



Nous avons appliqué le pattern command aux déplacements que doivent faire les éléments du jeu (move). Pour cela, nous l'avons implémenté comme sur le diagramme de classe ci-dessous.

3.2 Singleton

Nous avons utilisé ce pattern pour la classe Orca qui est une classe Creature. Cela signifie qu'il ne pourra y avoir qu'une seule instance de cette classe dans le jeu.

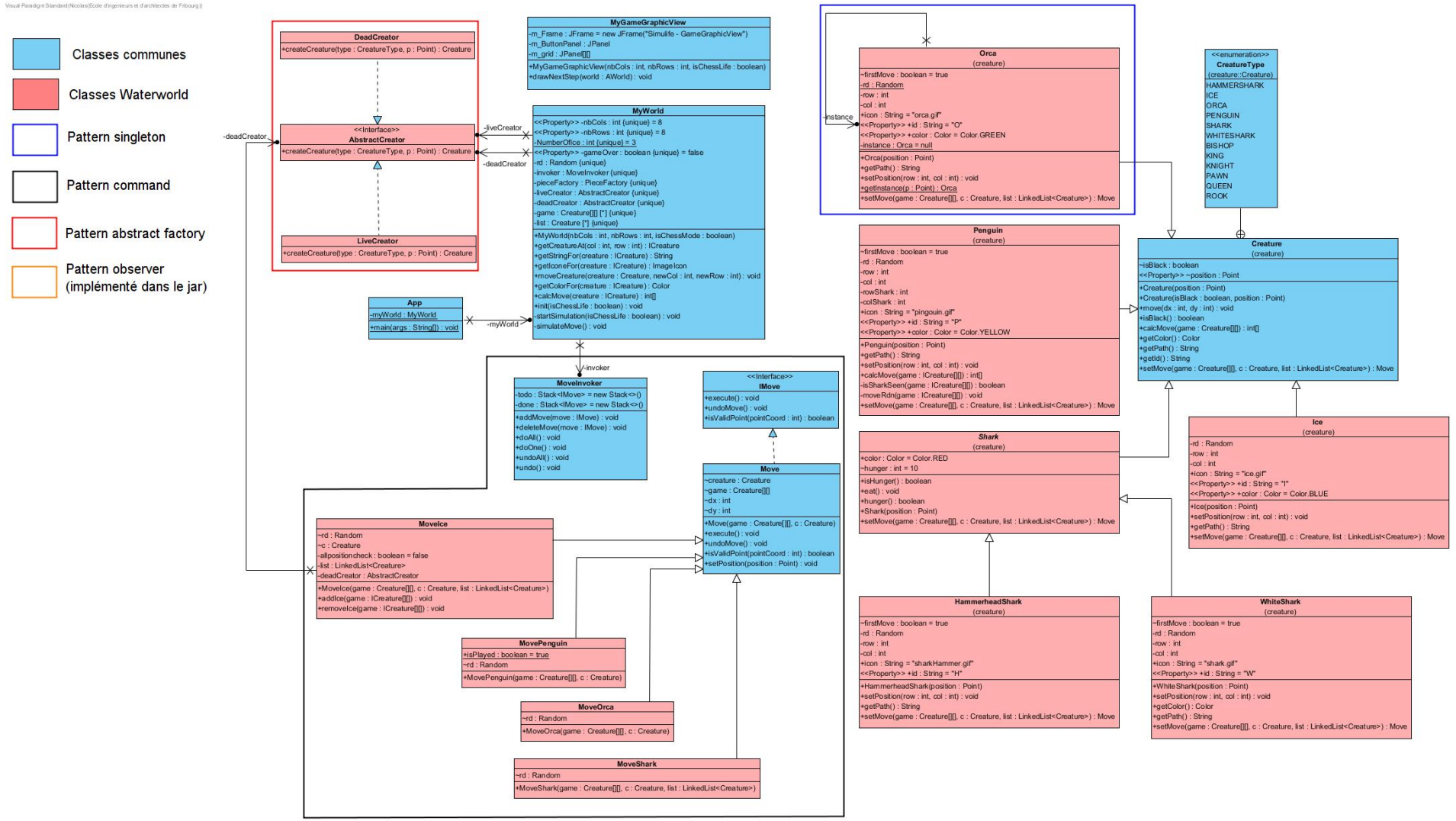
3.3 Observable

Ce pattern est utilisé pour la visualisation du jeu et il est implémenté dans les classes contenues dans le JAR.

3.4 Factory

Ce pattern est représenté en jaune sur le diagramme de classe. Il est utilisé pour la création des différents objets du système, dans notre cas des créatures (Creature). Il sert à séparer la partie création de la partie utilisation des créatures.

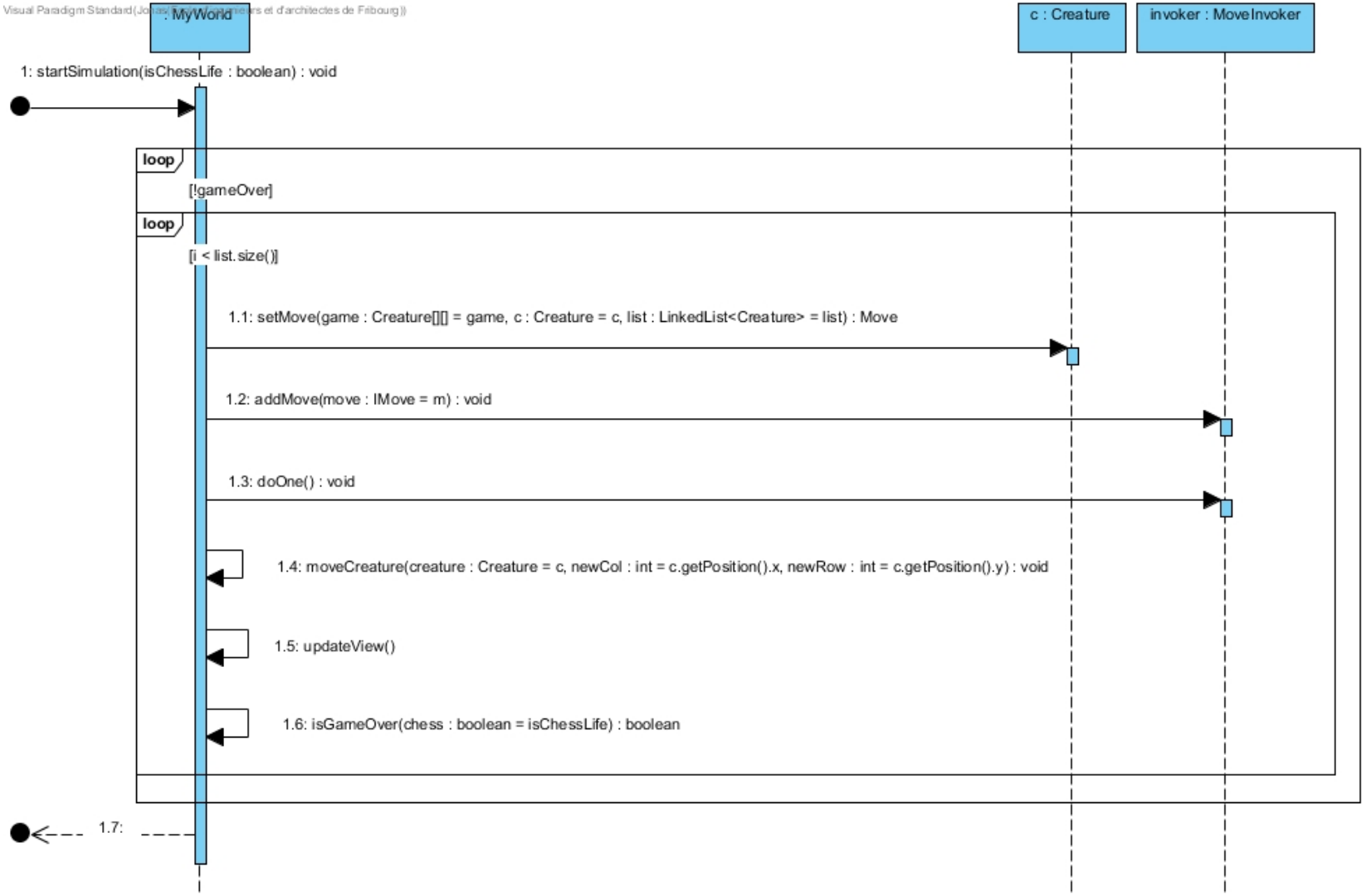
4 Diagramme de classe final après fusion



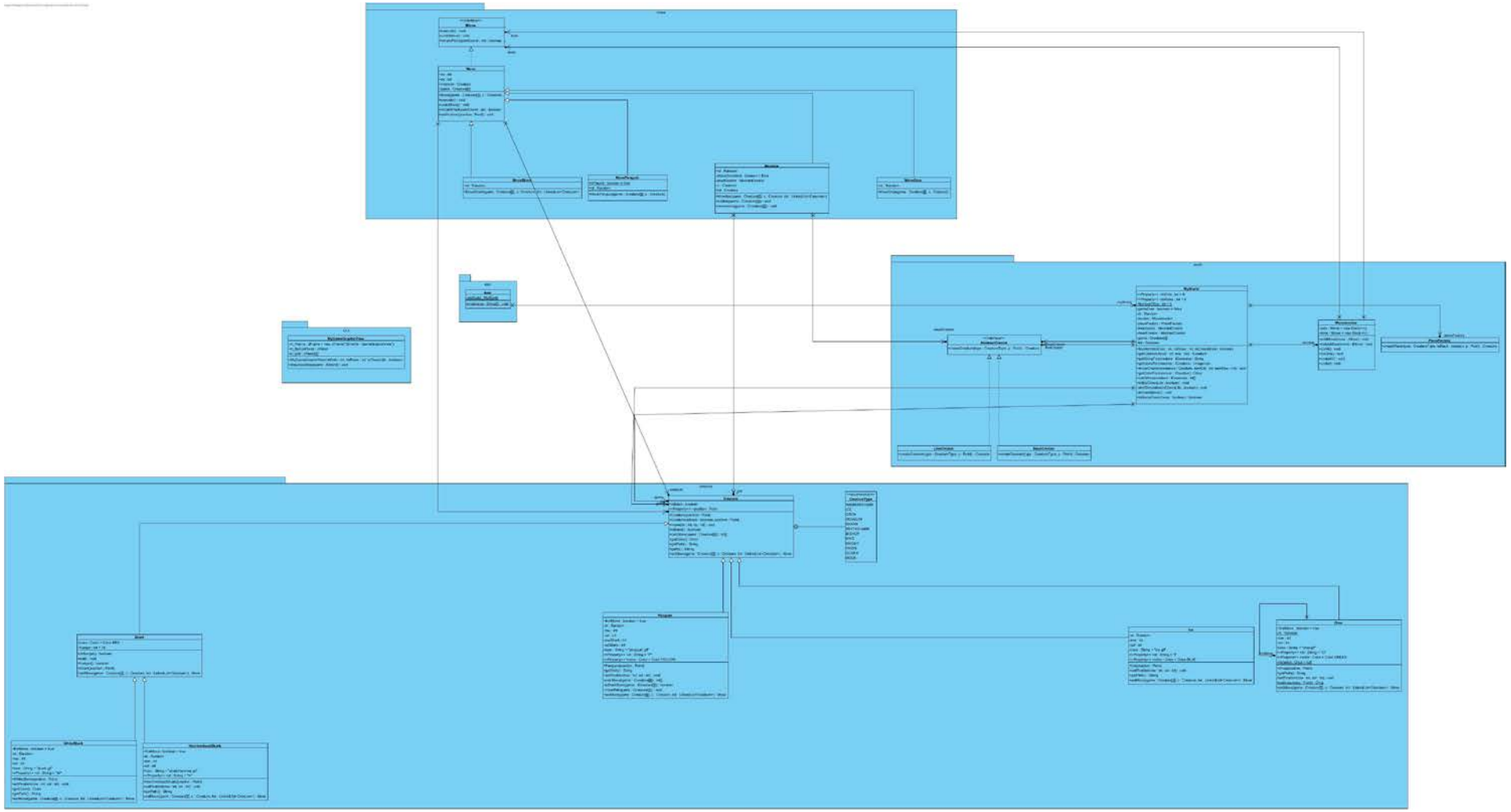
5 Modification diagramme de classe pour fusion

Pour procéder à la fusion, nous avons dû avoir une meilleure séparation au niveau des classes MyWorld et App. La création d'une classe Creature qui hérite de ICreature pour permettre une séparation des différents attributs et méthodes communs à toutes nos créatures. Avant la fusion, le pattern n'était pas implémenté correctement. Après la fusion, son implémentation a été rectifiée et améliorée.

6 Diagramme de séquence après implémentation



7 Diagramme de classe



8 Comparaisons/commentaires

Sur le schéma après fusion, on peut constater que la clarté du schéma permet de mettre en évidence les différents patterns. Dans ce schéma, il manque les différentes utilisations des classes.

Le diagramme (reverse) manque de clarté. Il ne permet pas de voir les différents patterns utilisés sur ce projet. Les différentes utilisations de nos classes sont représentées sur ce diagramme.

9 Analyse personnelle de l'utilisation des patterns

9.1 Command

Le pattern command nous permet de faire que la classe principal MyWorld ne connait pas les différentes Classes de mouvements. Cela permet de faire que la classe principale appelle des ICommand pour bouger toutes les créatures.
Cela permet de changer rapidement de comportement de mouvement d'une créature. Cette implémentation permet de pouvoir ajouter facilement une nouvelle créature avec un mouvement personnalisé.
Ce pattern est donc très intéressant pour le développement de notre projet.

9.2 Singleton

La classe est utile car nous avons seulement une créature Orca. Dans notre cas, il n'est pas nécessaire de construire un singleton pour une créature unique. Cependant, ce pattern est très facilement implémenté.

9.3 Observable

Le pattern observable est directement utilisé dans la librairie fournie. Il permet de manière simple de mettre en place un système de souscription pour les différentes vues. Ce pattern est très utilisé généralement pour la partie visuelle des applications.

9.4 AbstractFactory

Le pattern AbstractFactory permet d'avoir plusieurs fabriques concrètes. Il nous a permis d'avoir un factory pour le waterworld et un factory pour chesslife. Cela nous a aidé car il nous a suffit d'ajouter une fabrique concrète simplement. Ce pattern est fortement utile pour notre projet.

9.5 Factory

Le pattern factory est utilisé par le pattern AbstractFactory. Ce qui nous a été fortement utile. Donc il a un fort intérêt dans ce projet.

10 Pattern builder et bridge

Le pattern bridge permet de découper une interface d'une classe et son implémentation. Il nous serait utile si une partie du projet se trouvait sur un serveur ou à distance. Mais sinon, il n'est pas nécessaire d'intégrer ce pattern. Il pourrait être utilisé pour les différents mouvements.

Le pattern builder permet de faire un modèle de conception pour la création d'une variété d'objets complexes à partir d'un objet source. Il peut être utilisé pour les créatures.