LEMEUNIER Clement 11103680

ZAFIFOMENDRAHA Gabriello 11311399

M1if01

Gestion de Projet et Génie Logiciel

MOY Mathieu
2018-2019

Refonte du projet Poneymon_fx

MVC - Design Patterns – JavaFX

Le rapport comprend une présentation globale du projet, une motivation des choix d'architecture (et des patterns choisis), et leur explication en s'aidant de diagrammes.

PATTERNS GRASP

Modifications du projet de base :

L'application de départ ne possèdait que 3 classes. On voit bien qu'il y a plus de rôles que ça. Les rôles étaient répartis de la manière suivante :

App.java
- lancer l'appli
- créer le terrain de jeu
- ajouter la scène à la fenêtre et l'afficher

Field.java
- création des poneys
- création d'un tableau pour tracer les évènements, capturer le focus
- création du canvas
- initialisation du terrain de jeu
- boucle principale du jeu

Poney.java
- gérer la position et la vitesse
- initialisation des poneys (position et vitesse)
- création, chargement, affichage des images

On peut éventuellement commencer à déplacer des éléments dans d'autres fichiers, mais on se rend vite compte qu'il faut tout refactorer. L'affectation des responsabilités aux objets est mal réalisée. Field crée des instances alors qu'il ne remplit pas les règles à respecter pour effectuer ceci.

En ce qui concerne la dépendance, l'impact des changements et la réusabilité, on se rend compte que ce n'est pas optimal. Si l'on veut apporter une modification, il faut réécrire beaucoup de code. La complexité est beaucoup trop élevée, les objets sont difficilement compréhensibles et gérables en raison de leur couplage trop élevé.

On a donc voulu directement penser au MVC et, surtout, à une totale refonte du projet.

Les objectifs étaient les suivants : respecter le faible couplage, une forte cohésion, éviter que les variations d'un objet aient un impact sur d'autres éléments, gérer des alternatives dépendantes de type, et gérer le polymorphisme.

MVC

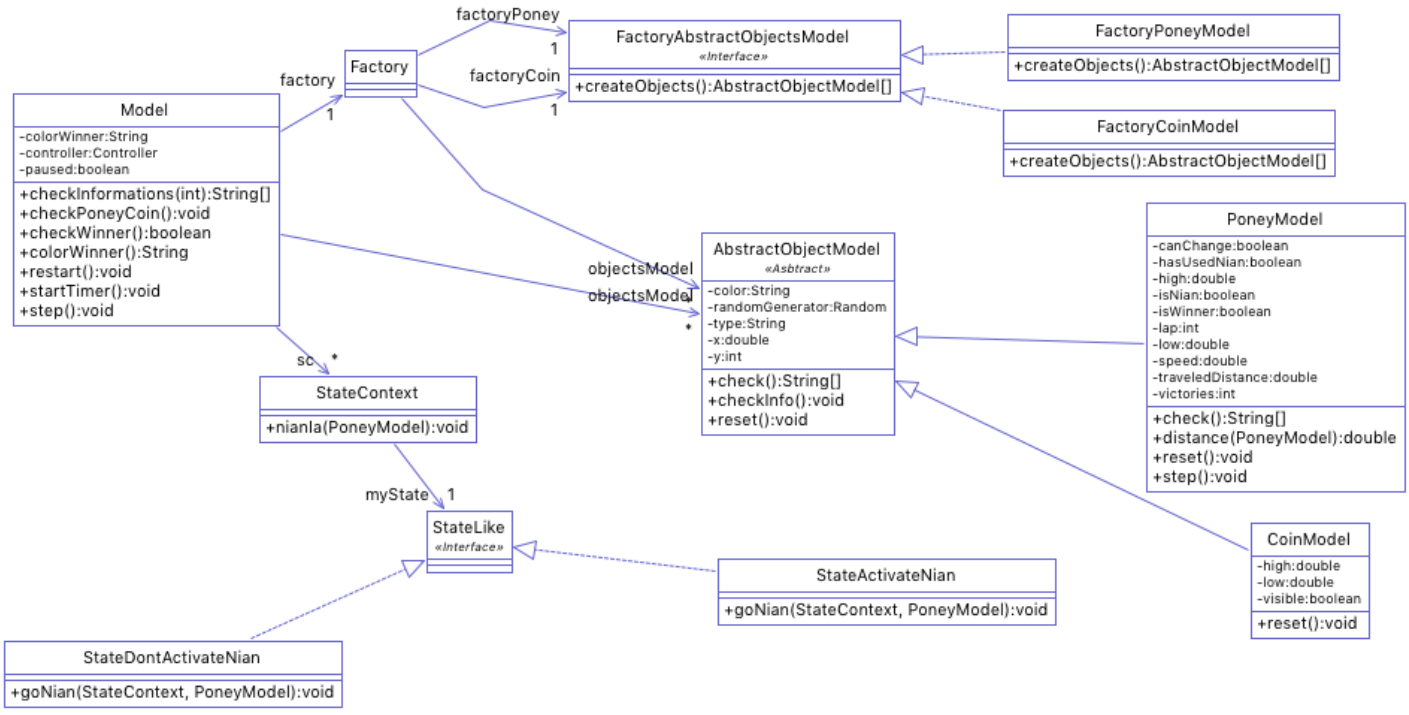
Le MVC était obligatoire. On a donc forcément les 3 classes suivantes : Controller.java, Model.java, View.java. Le programme a donc 4 packages : App, Controller, Model et View.

Comme on est susceptible d'avoir plusieurs vues, on a décidé d'avoir une classe abstraite AbstractView. Cette classe abstraite possède les éléments qu'ont en commun toutes les vues, c'est à dire un Group, une Scene, et une Stage.
Le contrôleur doit avoir une référence vers le modèle, et vers une liste d'AbstractView.
Le modèle doit en avoir seulement une vers le contrôleur.
Les AbstractView en ont seulement une vers le modèle.

L'App.java se charge de créer :

- une unique instance du contrôleur (singleton)

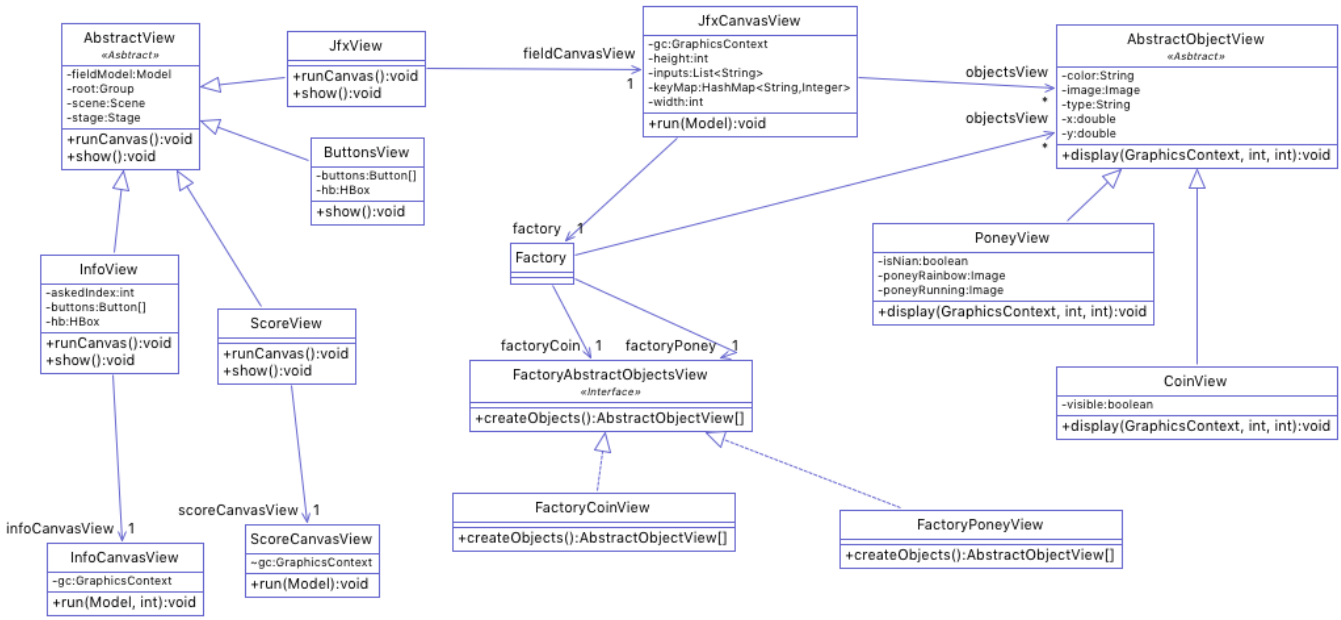
- une unique instance du modèle (singleton)
X est ici le nombre de poneys. Le modèle crée un tableau de X StateContext. Ces context servent à gérer l'IA (qui sera expliquée en détails plus tard).
Le modèle possède une liste d'AbstractObjectModel qui correspond à tous les objets du modèle (pour l'instant poneys et pièces, possibilité d'en ajouter d'autres dans le futur en créant une classe correspondant à ce nouvel objet). La création de ces objets se fait à l'aide d'une instance unique de Factory (détails sur cette factory plus tard).



- les vues
Les vues sont une extension de AbstractView.
Le contrôleur n'a donc plus qu'à appeler runCanvas() (qui sert à appeler la fonction run du canvas de cette vue, s'il elle en possède un) ou encore show() sur toutes ces vues.
Si la vue possède un canvas sur lequel elle doit afficher des informations, on lui associe une classe qui dérive de canvas, c'est le cas pour JfxView/JfxCanvasView, InfoView/InfoCanvasView et ScoreView/ScoreCanvasView.
Le canvas JfxCanvas possède une liste d'AbstractObjectView. Cette classe abstraite permet d'appeler la fonction getValuesFromModel sur les AbstractObjectModel correspondants.

Ici, la manière de récupérer les valeurs est la suivante :
Lorsque la liste d'AbstractObjectModel est créée, les objets sont en fait rangés dans cette liste dans un ordre précis, c'est à dire les poneys d'abord, en sachant que le premier est en fait celui de la première ligne, et le dernier poney est celui de la dernière ligne. Pour ce qui est des pièces, elles sont rangées en fin de liste, en suivant le même principe (la première est celle de la première ligne, la dernière celle de la dernière ligne). Il suffit donc, pour retrouver les pièces, d'itérer à partir de X (nombre de poneys), jusqu'à la fin de cette liste. Si on veut ajouter un troisième type d'objets, il suffit d'itérer de, premièrement, 0 à X-1, puis de X à 2X-1, puis de 2X à 3X-1, en faisant attention à l'odre dans lequel les objets sont insérés dans la liste. Cette méthode n'est peut-être pas très courante, mais cela nous a permis de pouvoir accéder facilement aux objets que l'on désirait dans cette liste.

On ajoute à ceci une vue ButtonsView, qui elle ne possède pas de canvas. Elle ne servira qu'à pouvoir appuyer sur un bouton correspondant à une couleur, et qui permettra d'activer le boost du poney de cette couleur.



App.java lance enfin la fonction principale du contrôleur run() qui se charge de démarrer le timer du modèle et de faire le lien entre modèle et vues.

Le modèle charge donc de lancer le timer, qui est appelé à chaque frame.

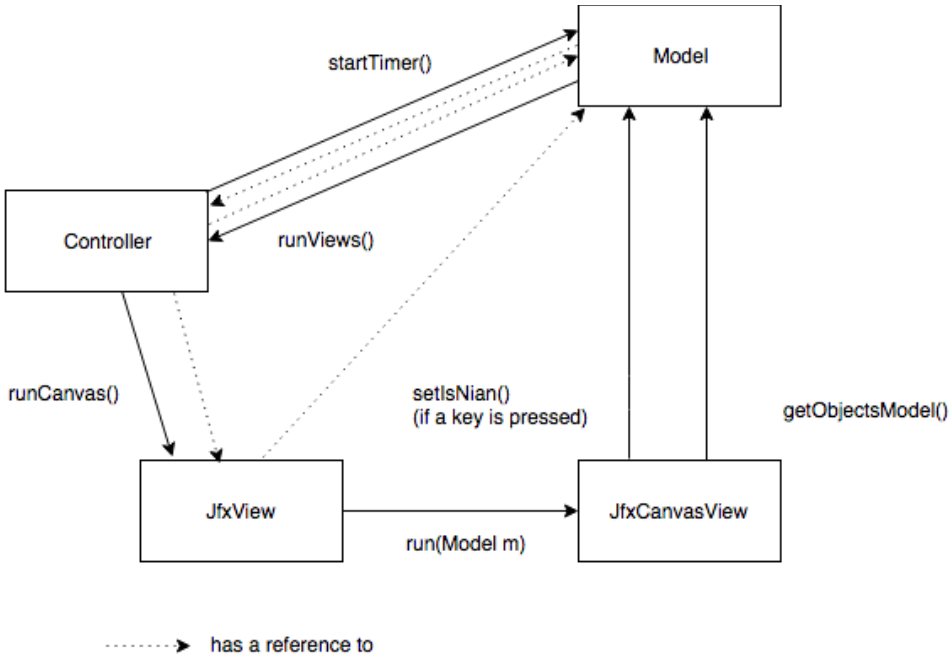
Le déroulement de ce dernier est décrit ici.
On appelle tous les éléments de la logique du jeu :
- step(), qui comprend checkPoneyCoin(), qui fait disparaître les pièces lorsqu'un poney la dépasse, la fait réapparaître ou non avec une chance de 1/5 lorsque le poney recommence un tour, etc.
- checkWinner(), qui vérifie si un poney a gagné.

Puis on appelle le contrôleur pour qu'il lance les fonctions principales de ses vues.

Viennent donc les vues :
Si la vue a un canvas à afficher, alors sa fonction principale appelle la fonction principale de ce canvas, qui comprend le remplissage du canvas, la notification du modèle si une touche a été appuyée, la récupération des valeurs du modèle, et l'affichage de tous les objects.

On revient donc au controleur, qui appelle enfin les fonctions show() de toutes ses vues. La fonction show() ajoute le canvas, s'il y en a un, ou les boutons, s'il y en a, au Group root de la vue.

Les interactions se font donc de la manière suivante :



DESIGN PATTERNS MVC appliqués

Singleton :

Nous avons décidé d'implémenter ce pattern car il correspond à l'état du contrôleur, du modèle et de la factory.
On s'est donc assuré que ces classes étaient des singletons.
Pour ce faire, on rend le constructeur de cette classe privé. On donne à la classe un membre INSTANCE de type cette classe. Le constructeur n'est accessible que par une méthode static de la classe, qui vérifie qu'il n'y a qu'une seule instance de créée.
On récupère donc ce singleton en appelant la méthode getInstance() de la classe.

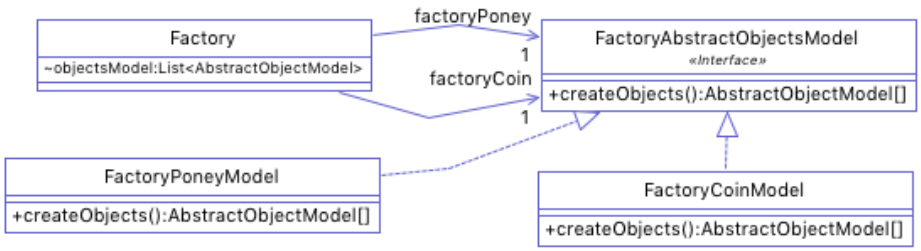
```java
public final class Controller {
    private Model model;
    private List<AbstractView> views = new ArrayList<AbstractView>();
    private static Controller INSTANCE;

    /**
     * Creates the controler.
     */
    private Controller() {
    }

    /**
     * Gets the only Controller's instance.
     *
     * @return INSTANCE.
     */
    public static Controller getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Controller();
        }
        return INSTANCE;
    }
```

Factory :

Nous avons décidé d'implémenter ce pattern car on a trouvé nécessaire de séparer le rôle de création des objets du modèle. De plus, il semblait correspondre au pattern Singleton.
Création d'une classe Factory qui instancie tous les objets nécessaires au modèle.
On a donc créé une interface FactoryAbstractObjectsModel qui ne possède qu'une méthode, qui renvoie un tableau d'AbstractObjectModel. On a ensuite créé deux classes qui implémentent cette interface, et qui se charge de la création, soit des poneys, soit des pièces. La classe Factory se charge donc d'appeler ces factories de poneys et pièces, et envoie un tableau d'AbstractObjectModel au modèle.
Le même système est appliqué à la JfxView pour la liste d'AbstractObjectView.



Etat :

Ce pattern semblait adéquat au développement d'un intelligence artificelle. Nous l'avons donc choisi afin d'implémenter un bon fonctionnement de l'IA.
Le modèle possède un tableau de StateContext de taille X (nombre de poneys). Ces StateContext ont un membre privé StateLike, qui est une interface possédant une méthode goNian(). Ce StateContext est initialisé avec un StateLike StateDontActivateNian. Quand le poney est dans cet état, on peut lui ajouter plusieurs conditions afin qu'il passe dans l'état StateActivateNian. Lorsque le poney est dans l'état ActivateNian, on active le mode nian.