Ambrosino Gwenaël 11212798
Auffret Quentin 11508216

Rapport

TP Réingénierie et design patterns en JAVA

Présentation du projet :

Le but de ce projet était de nous initier aux technologies des design pattern, pour cela nous avons réimplémenter le code du TP sur les poneys selon une architecture MVC.
Dans la première partie du projet nous avons donc travailler sur la réingénierie du code.
Dans la deuxième partie nous avons ajouté des éléments au jeu, tel qu'une interface, des obstacles ou une fonction de saut.

Choix d'architecture :

Pour la réingénierie du code nous avons donc implémenté un pattern MVC, son principe et de séparer le code en 3 parties, le modèle, la vue et le contrôleur :
-le modèle comprend tout les données de jeu, les poneys, les obstacles, leurs positions, les IA...
-la vue gère l'affichage du jeu et son interface
-le controlleur lui gère l'initialisation du jeu, les fonctions du modèle et de la vue en plus de cela

Dans notre cas, nous sommes confrontés à des classes changeant régulièrement de valeur, par exemple les poneys changent régulièrement de position et doivent envoyer de nouvelles informations à la Vue.
Deux implémentations sont possibles, la Vue se charge de demander les valeurs au modèle, soit le modèle se charge de les envoyer à la Vue dès qu'il subit un changement.
Nous sommes partis sur la deuxième solution car les poneys changeant constamment d'état, on ne peut pas se permettre de juste demander au modèle l'état actuel du poney car nous risquons de rater beaucoup de changement.

Nous avons donc ajouté un pattern observer, il définit une relation entre nos classes de type 1 à n.
Ainsi tout les objets en dépendant seront notifiés d'un changement d'état et mis à jour.
Celui-ci fonctionne avec deux interfaces, une interface observable et une observer.

Dans notre cas, le modèle le notifiera dès que celui-ci subira un changement, et les nouvelles données seront récupérées par la vue pour ainsi gérer l'affichage.

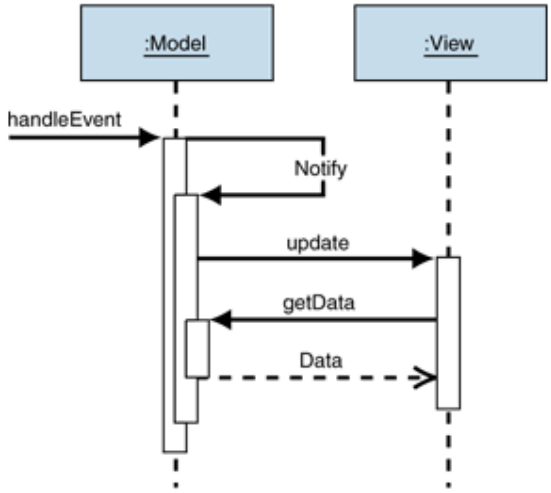


Diagramme de séquence représentant les échanges entre le Modèle et la Vue, l'observer notifie un changement, l'envoie à la Vue qui récupère la donnée pour gérer l'affichage.

La boucle de jeu se trouve dans le contrôleur, elle est gérée par la fonction AnimationTimer( ), on y gère le déplacement, le choix de la stratégie à appliquer pour chaque poney ainsi que la création d'obstacles.
Le contrôleur initialise également game et gameDisplay qui gèreront respectivement l'instanciation des objets dans le modèle et la vue.
La classe game fait actuellement le lien entre le terrain (la classe field) et le contrôleur, elle est aussi la pour accueillir des données générales de jeu s'il y a besoin.
La classe Field initialise le terrain, la création des poneys et la gestion des obstacles.

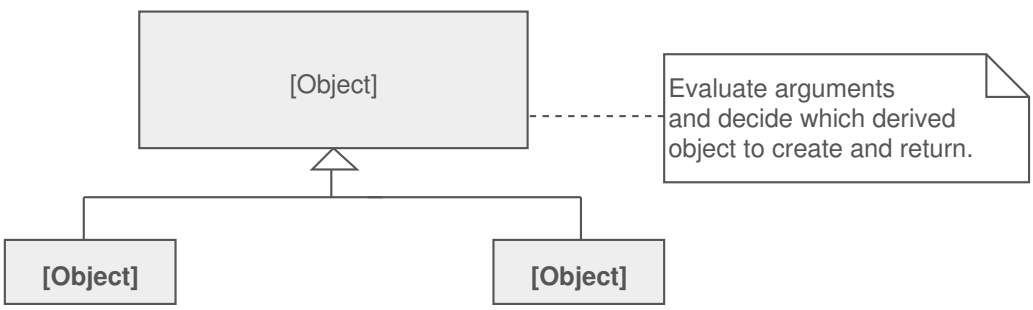
Du côté de la Vue, le fonctionnement en cascade est similaire, gameDisplay crée le fieldDisplay et lui même crée les runnerDisplays et les obstacleDisplay.

Design pattern :

-Constructeur :

Factory :

Le pattern factory nous permet l'implémentation d'une interface qui permet de choisir entre plusieurs sous classes, nous l'avons donc choisit pour la création de nos runners (classes principales à partir de laquelle on dérive les poneys et les nyanponeys) ainsi que pour la création des obstacles (dont on dérive les obstacles statiques et dynamiques).



-Utilisation pour la création des runners

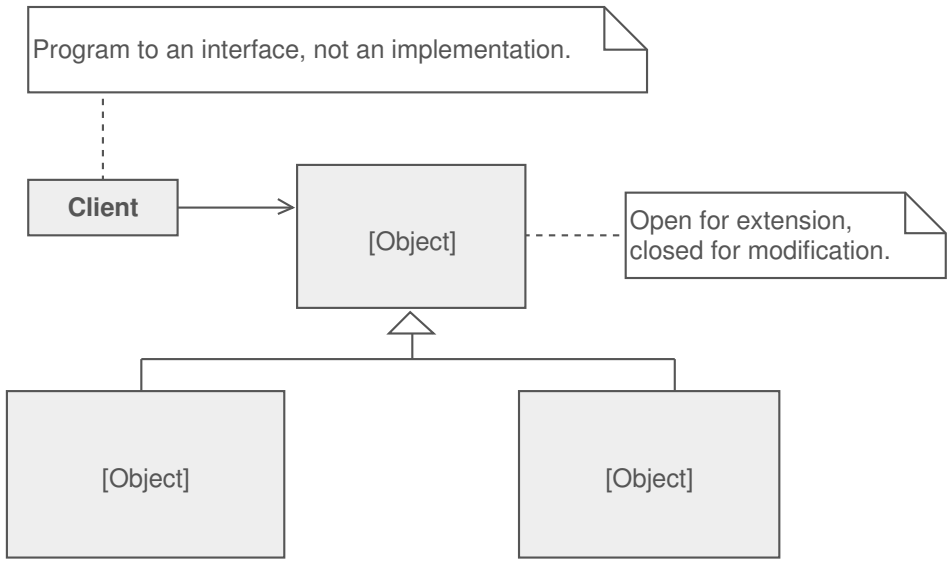
-Utilisation pour la création des obstacles

Avec du recul, nous aurions pu créer les obstacles en les dérivant de la classe runner, ainsi nous aurions eu seulement une factory à utiliser et seulement la fonction move à surcharger dans obstacle.

-Comportementaux :

Le pattern strategy encapsule différents algorithmes, on peut ensuite changer entre chaque algorithme en fonction du comportement voulu pour nos poneys.

Strategy :



On implémente une interface qui permettra de choisir entre chaque algorithme, celle ci contient une fonction qu'on surcharge dans les les classes dérivées et auquel on passera nos poneys en paramètre. Ainsi, suivant les conditions, l'interface nous permettra de choisir l'algorithme voulu. Le pattern stratégie nous permet donc d'implémenter un polymorphisme, dans notre cas, il sera dynamique, les runners connaissant notre interface directement.

-Utilisation pour le choix de la strategy à utiliser pour chaque poney (IA ou Humain)
-Utilisation pour le choix de la stratégie dans le choix du saut (IA ou Humain)
->Dans ce cas, on vérifie par exemple les conditions et les entrées clavier (dans le cas de l'humain)

Diagramme UML (réalisé avec Violet UML Editor 2.1.0.) :