Contents

Projet hex du Groupe 203

Membres : Yousrah SOULE, Aaron RANDRETH, Ilo RABIARIVELO, Adrien DEU **Groupe** : 203

Résumé de notre implémentation

Dans ce projet nous avons implémenté le jeu Hex, avec un interface sur la ligne de commande, et deux modes de jeux: "joueur" où deux joueurs s'affrontent, et "seul" où un seul joueur fait face à un ordinateur. Cette ordinateur n'a ici pas d'algorithme complexe, mais comme nous le montrerons dans ce rapport, il est très simple grâce à notre architecture de créer un robot plus complexe.

Notre structure de projet

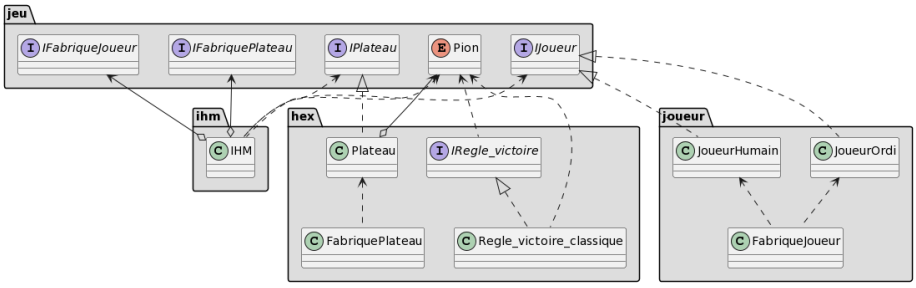
Quelques points à noter

Pour le jeu de hex, nous avons repéré deux axes de changements principaux: les joueurs, et le plateau. Pour les joueurs, notre jeu simple contient deux types de joueurs, mais on peut très bien imaginer des expansions qui créeraient des robots plus intelligents, ou des joueurs humains qui jouent un coup aléatoire tous les n coups. Pour le plateau nous avons utilisé un tableau à deux dimensions mais il est tout à fait possible que dans le future qu'une implémentation avec une structure de donnée différente soit requise.

Nous avons donc créé des interfaces et des fabriques pour ces deux types de classes. Afin de limiter le plus possible la dépendance de l'ihm, qui évolue plutôt lentement, à ceux-ci.

Nous avons utilisé le pattern de délégation sur notre classe Plateau, afin de nous permettre de facilement changer les règles du jeu, qui pourraient évoluer à un rythme différent de l'implémentation de plateau.

le diagramme UML



Ce que l'on pourrais rajouter

- Grace à la l'interface IRegle_victoire, sans modifier le reste du code, nous pouvons modifier les règles du jeux en créant une nouvelle classe qui l'implémente. Cette classe pourrait par exemple faire gagner un joueur si il lie deux côté adjacent ou transformer les cases hexagonales en carré.
- En créant de nouvelles classes joueurs puis en modifiant la fabrique, nous pouvons augmenter la difficulté du mode "seul" en faisant un robot intelligent.
- En créant de nouvelles classes plateau puis en modifiant la fabrique, nous pourrions rendre le jeu plus rapide mais plus gourmand en espace, ou vice-versa
- En modifiant uniquement le switch case de la classe IHM nous pouvons créer d'autres modes de jeu, comme un mode robot contre robot.
- Comme l'IHM est isolé de toutes les autres classes, pour rajouter d'autres types d'IHM comme un GUI il suffit de créer une classe sans modifier le reste du programme.

Synthèse de nos tests unitaires

Pour les joueurs nous avons des tests de cohérence assez simple pour s'assurer que:

- Les noms sont ceux attendus; le nom donné pour l'humain, et le - nom séquentiel (R0, R1..) pour le robot.
- Les joueurs jouent bien au bon endroit

Pour le plateau nous avons vérifié que:

- Le plateau s'affiche comme voulu.
- La fonctionnalité de recréation d'un plateau à partir d'un état de jeu déjà commencé fonctionne.
- La validité ou la possibilité de placer un pion sur le plateau.

- Agagne fonctionne pour tout les joueurs, dans le plus de scénarios divers possibles pour essayer d'éviter les cas spéciaux.

Bilan du projet

Nos difficultés

Un des points sur lequel nous avons passé le plus de temps sur le projet a été la gestion des dépendances. Dès le début avec les différents packages nous avons créé des interfaces, pour inverser les dépendances et éviter à l'IHM d'être dépendant des classes concrètes. Cependant, à cause de la différence de comportement entre le joueur humain qui a besoin d'un dialogue de l'IHM et le joueur robot qui n'en a pas besoin, nous avons finit par avoir des dépendances textuelles néfastes: comme illustré ci-dessous:

```
if (j1.getClass().getName().equals("joueur.JoueurHumain")) {

  System.out.println(j1.getNom() + " : Saisir les coordonnées du pion à poser
  sur le plateau :");

  String coord = sc.next();

  j1.jouer(coord, plateau); }
```

Pour régler cela nous avons rajouté une méthode dans l'interface IJoueur:

```
public boolean needs_input();
```

qui nous permet de généraliser le code et d'éviter de rendre l'IHM dépendant à une implémentation spécifique:

```
  if (j1.needs_input())
    // saisie des coordonées par le joeur

  j1.jouer()
```

Un aspect que nous n'avons pas réussi à résoudre a été les dépendances lié aux fabriques. Même si l'IHM n'est pas du tout dépendante des différentes classes Joueurs, et Plateau, elle est dépendante des classes concrètes des fabriques. Cette dépendance étant cependant une dépendance de création, à un endroit très simple à modifier, elle n'est pas énormément néfaste.

Les deux attributs de IHM, où se trouve cette dépendance:

```
    private static IFabriquePlateau fplateau = new FabriquePlateau();
    private static IFabriqueJoueur fjoueur = new FabriqueJoueur();
```

Ce que le projet nous a appris

Avec ce projet, durant lequel nous avons fortement utilisé github, toute l'équipe s'est amélioré à l'utilisation de git que ce soit avec l'interface web de github ou avec la ligne de commande.

En terme de programmation, le projet nous a permis de mettre en pratique les principes SOLID, et les Designs Patterns que l'on a étudié en TP. Nous avons donc pu observer en utilisant ces outils la facilité avec laquelle on peut modifier ce jeu par rapport à nos projets antécédents.

Le résultat final

Nous avons été plutôt satisfait par le projet, il nous semble très flexible et facile à faire évoluer, et toutes les fonctionnalités que nous voulions implémenter fonctionnent.