Projet de programmation orientée objet et d'interfaces graphiques

Julien COOLEN Yaniv BENICHOU

Année 2018 - 3e semestre L2

1 Introduction

Nous abordons dans ce rapport nos choix de modélisation et les difficultés rencontrées au cours de ce projet pour le cours de programmation orientée objet et d'interfaces graphiques.

Le but de ce projet consiste à implémenter plusieurs jeux de plateau en regroupant les fonctionnalités communes. Pour cela nous avons développé une plateforme logicielle [1] à partir de laquelle nous avons codé chaque jeu : un ensemble d'objets qui intéragissent entre eux et permettent de développer des jeux de plateau dans la plus grande généralité. Ce framework devait être facilement étendu pour implémenter un nouveau jeu.

Aussitot le projet entamé, nous avons mis en place un dépot gitlab afin de collaborer au mieux à deux , en ayant un accès aux dernieres versions de codes mises à jour par l'un d'entre nous, mais aussi avoir une sauvegarde des états du projet au fur et à mesure de notre avancée. Nous avons codé tous les jeux demandés : dominos que l'on assemble pour former une chaîne linéaire, dominos gomettes que l'on assemble pour former une chaîne avec des branches (dans le plan), un puzzle ainsi que le jeu du saboteur qui est un veritable jeu de société demandant plus de strategie de la part des joueurs.

2 Modélisation

Nous avons tout d'abord remarqué que chaque jeu s'articule toujours autour des mêmes objets : un plateau, des pièces, des cartes, une pioche, des joueurs et leur main. Ce sont les composants de base de notre framework. Chacun de ces objets est éventuellement étendu pour ajouter ou redéfinir des comportements.

Chaque jeu adopte la structure imposée par le framework.

1. En anglais *framework*.

Nous nous sommes toujours questionné sur la manière dont on souhaite agencer les pièces, les données que ça implique, ce que l'on souhaite pouvoir faire, dans quelles conditions, quel impact ça aurait sur le reste, etc.

Nous avons de plus essaye d'exploiter au maximum les concepts étudiés ainsi que des evenements pour interagir avec la vue.

2.1 Présentation du framework

Le framework définit les types de base ainsi que leurs interactions que chaque nouveau jeu peut composer. Nous avons une interface générique plateau `Board<T extends Tile>` qui manipule des pièces posables (interface `Tile<S extends Side>`). Des méthodes très générales permettent de déposer des pièces sur le plateau, récupérer une pièce à une coordonnée précise (en introduisant une classe `Coordinate` qui permet de manipuler des coordonnées, en les additionnant, etc... et des exceptions si l'on donne une coordonnée en dehors du plateau que l'on capture à un plus haut niveau d'abstraction, dans la boucle des jeux) ou encore une méthode de parcours en largeur de graphe `hasPathFromTo(Coordinate start, Function<T, Boolean> isGoal)` qui parcours les pièces adjacentes jusqu'à une pièce vérifiant une certaine condition `isGoal` utilisée par le jeu du saboteur.

Ces interfaces sont accompagnées d'une implémentation par défaut, par exemple l'interface tuile `Tile<S extends Side>` possède une méthode très générale `boolean fitsWith(Tile<S> t, Direction d, SidesMatch<S> matchRule)` qui retourne vrai si deux tuiles s'assemblent en fonction d'une règle définie par l'interface fonctionnelle `SidesMatch<S extends Side>` qui vérifie si deux côtés peuvent s'assembler :

```
@Override
public boolean fitsWith(Tile<S> t, Direction d, SidesMatch<S> matchRule) {
    return (t != null) && matchRule.apply(getSide(d), t.getSide(d.getOppositeDirection()));
}
```

Nous avons également définit les types `Deck<C>` et `Hand<C>` pour la pioche et la main d'un joueur avec une implémentation par défaut de leurs méthodes comme par exemple `void distributeCards(List<? extends Hand<C>> hands) throws EmptyDeckException` pour distribuer des cartes à plusieurs mains ou encore `void deal(Hand<C> hand) throws EmptyDeckException` pour piocher une carte depuis le haut de la pile de la pioche.

Nous avons fait en grand usage de la généricité en ajoutant des bornes pour les types pour pouvoir s'assurer que l'on ne puisse pas déposer n'importe quel types d'objets sur une plateau par exemple. On implémente l'interface `Board` pour le jeu du saboteur comme suit : `class SaboteurBoard extends BoardImpl<SaboteurTile>`. De cette façon l'implémentation garanti que le plateau ne peut contenir qu'un seul type d'objet, les tuiles du saboteur et pas des tuiles des dominos ou autre, ce qui permet d'éviter ce type de bugs qui sont levés à la compilation et non à l'exécution.

Ainsi que quelques types énumérés pour les directions et les coordonnées relatives selon la direction (cf. package `common.enums`) pour un code plus robuste.

2.2 Dominos simples et avec gommettes

Nous avons commencé par implémenter le jeu des dominos simples à l'aide d'une `ArrayList`, les pièces étaient ajoutées aux extrémitées de la liste. Une méthode vérifie si une pièce peut se poser à gauche ou à droite d'une autre pièce. Lorsque nous avons implémenté le placement des pièces pour le jeu du saboteur, nous avons fait évoluer notre modèle en introduisant les types `Tile<S extends Side>` pour les pièces que l'on pose sur un plateau et `Side` pour les côtés des pièces, ce dernier type servant à regrouper les variables qui définissent un côté.

Nous avons en revanche constaté que le jeu des dominos simples peut être vu comme un sous-ensemble des dominos gommettes, dans le sens où les dominos simples s'assemblent pour former une seule chaîne contrairement aux dominos gommettes avec lesquels on peut en plus former des branches.

C'est pouquoi nous avons en plus implémenté le jeu des dominos simples et dominos gommettes en étendant une classe plus générale qui permet de placer côte à côte des pièces rectangulaires sur un plateau si leurs côtés se correspondent. Nous trouvons cette petite expérimentation intéressante.

2.3 Saboteur et puzzle

Après avoir codé le jeu des dominos simples nous avons réalisé qu'il serait plus intéressant d'implémenter le jeu du saboteur car son implementation representait l'implementation demandant le plus de reflexion et donc de temps de part sa complexité de jeu (verifier quatres cotés d'une carte ainsi que la connexion de ces chemins, plusieurs types de cartes permettant de bloquer/debloquer un autre joueur). Une fois écrit nous avons pu réutiliser beaucoup de code. C'est au cours du développement de ce jeu que nous avons développé une base de code commune réutilisable et facilement extensible pour chaque jeu. Les autres jeux etant toujours des jeux de plateaux, mais en une version simplifiée.

2.4 Pattern MVC

L'architecture modèle-vue-contrôleur à été implémentée pour les dominos simples et le jeu du saboteur. Pour cela nous avons introduit une

3 Difficultés rencontrées

Lors du développement nous avons réécrit à plusieurs reprises certaines méthodes ainsi que certains aspects de notre modélisation, ce qui introduisait quelques fois de nouveaux bugs. C'est pourquoi nous avons écrit des tests unitaires à l'aide de la librairie jUnit que l'on lance à chaque modification pour identifier toute régression dans le code.

l'implementation du jeu du saboteur nous a demande un certain temps avant de parvenir au resultat exigé .L'utilisation d'Enums pour les cartes Path ainsi que l'utilisation de classes abstraites, d'interfaces et d'heritages ,nous a finalement permit de surmonter ces quelques (mais legers) obstacles.

Cependant, nous sommes d'accord pour admettre que la tache qui nous a paru la plus compliquée a été de mettre en place une interface graphique. En effet, malgrè que nous soyons habitués à utiliser les concepts d'heritage,d'interfaces,de classes abstraites,de programmation orientée objet,d'exceptions ou encore de généricité, nous n'avions que peu de connaissances en Interface graphiques.Finalement, la documentation Java et de nombreuses heures de tests successifs et de discussions dans des Forums, nous ont permit de vous presenter cette interface-ci, qui peut etre largement ameliorée.

Malgrès tout, il est evident que ces difficultés rencontrées n'ont été que bénéfique pour nous, nous ont permit d'en apprendre d'avantage sur l'utilisation de l'intégralité des concepts étudiées ce semestre en cours de Programmation Orientée Objet(et bien plus encore), et surtout sur l'implementation d'une interface Graphique.

4 Pistes d'amélioration

Pour ce qui est de l'interface graphique, nous avons essayé de permettre aux joueurs d'effectuer n'importe quelle action , en les laissant dans un premier temps ,cliquer sur la carte de leur main à jouer , puis dans un second, à cliquer sur la case ou joueur où effectuer l'action , ces derniers rémarqués par un changement de bordures (rouge). Tout ceci evidemment gerer par des ActionListeners. Malheureusement, après plusieurs essaies et tirages de cheveux pour bien limiter à 2 clicks (le premier dans la main et le second ailleurs), ainsi que de recuperer la carte au click associé, nous avons préféré une solution "plus simple" qui est de recuperer les valeurs à partir de pop-up, lancées par des fonctions display(), grace à l'utilisation de JOptionPane. Il est evident que cet echec est une grande frustation pour nous,et nous savons qu'avec un peu plus de temps, nous aurions trouvées le bon moyen d'y parvenir. Nous comptons chercher ce moyen après la date de rendue de ce projet pour notre apprentissage personnel.

5 Pistes supplementaires

Nous aurions apprecié implementer bien d'autres options supplémentaires comme par exemple , permettre à l'utilisateur d'effectuer des sauvegardes de parties puis de les charger, ou encore mettre en place des petites animations dans un background (gifs) mais faute de temps, nous nous sommes vue obligés de nous concentrer principalement, sur ce ce qui été exigé.

Nous avons pu tout de meme ajouter une piste audio (en boucle) lors de l'ouverture de l'interface graphique.

6 Conclusion

Nous avons développé le projet itérativement en travaillant à plusieurs niveaux d'abstraction : sur chaque jeu et le framework. Le développement du framework était guidé par les besoins de chaque jeu et réciproquement le développement d'un jeu dépendait directement de l'organisation du framework. Ainsi, si une extension du code était compliquée à écrire, nous revoyions le modèle. Cet aspect du projet à été de loin le plus intéressant et enrichissant puisqu'il nous a permit remettre questions nos choix, de les rectifier et perseverer.

Il était de plus très plaisant de remarquer que beaucoup de concepts se retrouvaient dans les jeux à implémenter, ce qui nous a conduit à introduire de nouveaux types qui ont grandement facilité le développement des jeux.

Nous avons également apprécié la liberté qui nous a été donnée au cours de ce projet quant aux choix de l'implémentation.

Le projet était long, la modélisation très intéressante, cependant par manque de temps nous n'avons pas pu consacrer autant de temps que nous aurions voulu aux interfaces graphiques.