Projet Programmation orienté Objet : Anti Monopoly

Rim GHERMAOUI — KIROUCHENASSAMY Badmavasan

Tuteurs : Bragagnolo Santiago — Pablo TESONE

1 Introduction

Le but de ce projet sera d'implémenter la simulation d'un jeu s'inspirant du Monopoly en langage java. Les joueurs ont pour objectif d'augmenter leurs patrimoines en parcourant un chemin circulaire contenant différentes cases où ils dovient prendre des décisions d'investissement ainsi que faire certaines actions. La simulation s'arrête si et seulement si l'une des trois conditions suivantes est atteinte :

- L'utilisateur demande à arrêter la partie (on demande à l'utilisateur s'il veut continuer ou s'il veut arreter de jouer).
- L'état a échoué (il ne possède plus de patrimoine).
- Il reste plus qu'un seul joueur dans le jeu.

Pour la suite, on suppose qu'il y a deux types de richesse, richesse en liquide et richesse en termes d'investissements.

2 Schéma UML

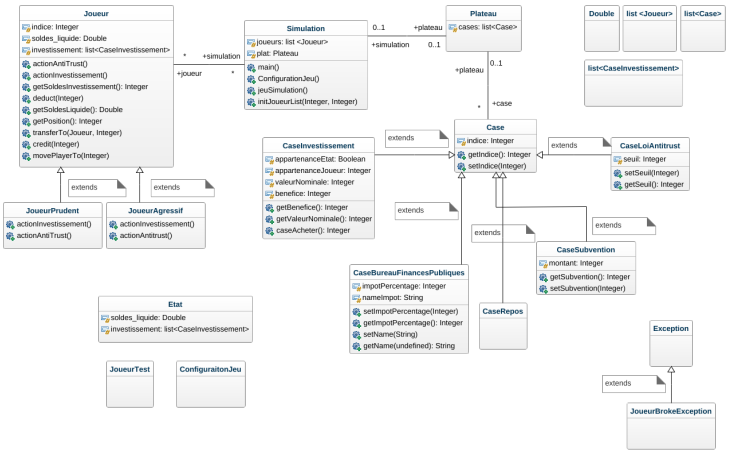

Figure 1: L'image orignale du fichier est disponible dans le dossier

3 Explication des classes et choix des structures de données

3.1 Joueurs

Dans notre classe jeu, nous avons une liste de joueurs, la structure de données sera du type ArrayList. Notre choix de structure de données s'est tourné vers celle-ci puisqu'à la fin du jeu, il faudra représenter les joueurs en fonction de leurs richesses respectives ce qui fait que nous serons amenés à utiliser le "sort". La collection List possède des méthodes propres à elle-même qui facilitent la façon de faire le sort. De plus, pour parcourir la liste, nous pouvons utiliser iterator de la collection ce qui nous a semblé plus simple et efficace.

Etant donné qu'il existe plusieurs types de joueurs, les classes correspondant aux types de joueurs vont être représentés par des sous classes de la classe Joueur. Cette classe contient des informations primordiales au déroulement du jeu tel que la manière dont le joueur va réagir face à deux des différentes types de cases à savoir "investissement" et "loi antitrust".

Les deux types de joueurs indiqués dans le sujet ne vont pas réagir de la même manière. Nous avons donc des fonctions abstraites dans la classe Joueur et ces fonctions présentent dans chacune des sous-classes correspondantes à chaque type de joueur qui définissent leurs façons de jouer.

3.2 Plateau

Pour cette classe-là, nous avons opté pour un array étant donné que nous n'aurons pas besoin des fonctionnalités du type sort ou autre qui nous amènerait à avoir besoin des différentes fonctionnalités de la collection. Notre choix ne s'est pas tourné vers l'ArrayList car cette dernière occuperait beaucoup d'espace mémoire sans forcément que ça ait un intérêt particulier. Nous avons donc préféré réduire la complexité au niveau du stockage de mémoire.

Plateau est donc un array de cases. Nous allons initialiser un plateau en dure. (On peut écrire un programme en fonction du nombre de cases subvention et le placement, mais comme c'est pas demandé et comme on va essayer de faire la simulation déjà pour qu'un seul plateau c'est à dire qu'un seul profil : le profil donné dans l'énoncé).

Pour construire le plateau, nous aurons également besoin des configurations de départ à savoir les sommes pour subvention / investissement, etc.

3.3 Case

Chaque case contient une action déterminée que le joueur doit faire. Ce sera donc une classe Case qui regroupe des attributs et les méthodes globales accompagnés des sous classe qui correspondent aux types de cases.

Dans le cas d'investissement : Si la la case investissement appartient à un joueur, le joueur en question doit payer une somme d'argent à l'autre. L'information sur le joueur x sera stockée à l'intérieur de la classe case. Si ça appartient à personne (ce qui correspond à si ça appartient à l'état), le joueur peut acheter.

Dans la case de subvention : une somme x sera créditée de l'état du joueur. La case contient l'information sur le montant à créditer.

Dans le cas de la case loi antitrust : le seuil x sera stocké dans la case. Le seuil correspond à la limite au dela de laquelle le joueur ne peut pas recevoir de l'argent. Si le montant que possède le jour est au-dessus de ce seuil, ce dernier est obligé de céder une partie. Dans ce cas, tout l'argent au dessus de ce seuil revient à l'état.

Dans le cas de Bureau Finances Publiques : une somme x (qui sera calculée à partir d'un pourcentage) sera transféré du joueur à l'état. La case contient la valeur de ce pourcentage (lors de

la configuration il faut faire attention, il faut demander à l'utilisateur de donner une valeur entre 1 et 100.

Dans le cas de Repos, il ne se passe rien puisque cette case correspond à une case dans laquelle aucune action n'est imposée.

Tous les cases contiennent des actions de get et set puisque nous utilisons des protected class ainsi que les packages. On utilise la classe case à l'intérieur du package casePackage dans la classe simulation qui se situe à l'intérieur du jeuPackage.

3.4 Configuration

La configuration est très utile pour la création du jeu. L'utilisateur doit pouvoir voir ces configurations s'il le souhaite. Donc ça serait une bonne idée d'implémenter une classe configuration qui contient toutes les configurations. Pour être plus organisé, on peut créer un package config et créer cette classe à l'intérieur de cette package. Donc il va falloir faire des get/set afin de mettre les attributs en private (pour ne pas permettre de modifier ces valeurs par n'importe quel utilisateur)

3.5 Exception

Il faut également des exceptions : une des exceptions qui nous permet de terminer le jeu est l'exception quand le joueur n'a pas assez d'argent pour payer. Dans notre cas, ca sera JoueurBrokeException.

3.6 Tests

Pour rendre le projet solide, il faut tester les méthodes d'une manière plus efficace.
Supposons qu'on a une fonction f qui utilise une fonction g. Ca serait inutile de faire deux tests séparés pour les deux fonctions. Il serait plus judicieux de faire des tests de manière intelligente et efficace. Soit on peut regrouper tous les tests dans un seul fichier sinon répartir les tests en fonction de leur contenu : par exemple une classe testJoueur qui test les méthodes du joueur et une autre type case qui test les méthodes de case. Cela est qu'une question d'organisation et taille du projet. La taille de ce projet Monopoly est petite donc c'est possible de garder tous les tests dans un seul fichier.

Les tests qu'il faut absolument faire :

- Création du plateau qui est primordial car sans le plateau on ne oeut pas faire tourner le jeu.
- Vérification que toutes les actions de toutes les cases se font de manière correcte.
- Vérfication si le jeu s'arrête si l'une des 3 conditions citées aupauvravant est atteinte.
- Vérification des actions des joueurs dans la case Anti-Trust et Investissement comme dit dans l'énoncé.
- Les fonctions élémentaires de Joueur, qui permettent de bouger sa position, créditer et débiter de son solde en liquide.
- Générations des exceptions.
- Vérification de si configuration est initialisé.
- Vérification de si la liste des joueurs est initialisée correctement c'est à dire avec le bon nombre de joueurs agressifs et prudents.

3.7 Simulation

Pour la simulation, on va initialiser une liste de joueurs qui sera du type ArrayList.
Mais une fois qu'il ont perdu, il faut trouver un moyen de le représenter. Il y a deux possibilités :

- Soit on ajoute un Boolean à l'intérieur de la classe joueur qui représente s'il a perdu ou pas. Dans ce cas, on sera obligés d'ajouter une condition qui vérifie si le joueur a déjà perdu ou pas à chaque tour. Et on sera quand meme obligé à parcourir tous les joueurs meme s'il y a que deux joueurs qui joue. Par exemple, on a au total 50 joueurs et il y a 48 qui ont perdu. 2 joueurs sont en train de joueur. C'est inutile de parcourir toute la liste surtout qu'en terme de coût de traitement, celà coûte trop cher.
- La deuxième proposition, aussi une proposition idéale, sera de créer deux listes. Une liste de joueurs qui joue et une liste de joueurs qui ont deja perdus. Cette dernière évite le parcours et les conditions de vérification inutiles. En terme de mémoire, on a le même nombre de joueurs donc dans notre implémentation, on va utiliser cette méthode.

4 Pseudo code de la simulation

```
Action Simulation :
    D: Plateau de classe plateau , Joueurs : ArrayList type Joueurs
    Tant que le jeu n'est pas arrete par l'une des trois conditions :
        Tant qu'un tour de tous les joueurs n'est pas fini :
            New position = Lance d
            Joueur avance a New position
            Joueur fait l'action indique dans la case
        Fin Tant que
    On demande si l'utilisateur souhaite continuer
    On v rifie si l' tat  n'a pas   chou
    On v rifie s'il reste plus qu'un joueur
    Fin Tantque
FinAction
```

5 Aspects Techinique du projet

On va utiliser éclipse IDE qui permet de compiler et exécuter les programmes à l'intérieur du logiciel même.
On va aussi utiliser git pour travailler à distance et faciliter la conception de l'application monopoly. De plus, pour tous les schémas UML , nous avons eu recours à genMyModel qui ne permet pas de générer automatiquement les classes de java mais qui pour autant permet de disposer d'une vision globale du jeu Anti Monopoly.

6 Conclusion

Pour conclure, nous avons conscience qu'il peut y avoir des tests supplémentaires qui dépenderont de l'implémentation du code. Nous allons probablement ajouter d'autres tests et exceptions si nous nous rendons compte que c'est nécessaire. Pour la suite, nous allons essayer d'implémenter notre idée avec une complexité minimale en terme de coup de traitement et mémoire.