PROJET INFORMATIQUE & STATISTIQUE

-

ANTI-MONOPOLY

RAPPORT FINAL

IS2A3 - Mercredi 23 juin 2021

Encadrants : Santiago BRAGAGNOLO
Pablo TESONE
Auteurs : Caroline SCHMID
Brinda TSOBGNI

Table des matières

Chapitre 1

Contexte du projet

Au cours de ce projet, on représente une partie de Monopoly dont la simulation est basée sur des joueurs pouvant être soit Prudents, soit Agressifs. L'un des joueur est l'État, il ne prend pas directement part au jeu mais en est un élément important.

De plus, la construction du jeu dépend de la configuration que l'utilisateur choisit en lançant le programme. Ce choix détermine les valeurs des cases comme par exemple la valeur de la taxe à payer, mais également les moyens fournis aux différents joueurs en lançant le jeu, car dans certaines configurations les joueurs ne partent pas d'un même montant.

Le plateau est constitué de cases de différents types, ces derniers déterminant les actions à réaliser en arrivant sur la case.

Les joueurs sont caractérisés par leur style de jeu.

La simulation crée les différentes entités nécessaires (un plateau, des joueurs, ...) et les fait jouer jusqu'à ce que soit l'État n'a plus de moyens, soit qu'il ne reste plus qu'un seul joueur sur le platau, voir aucun, ou encore que l'utilisateur décide de mettre fin à la partie.

Chapitre 2

Détail de la Simulation

La simulation peut être découpée en 3 étapes :

1. Initialisation du jeu
2. Déroulement du jeu
3. Résultats

2.1 Initialisation du jeu

On demande à l'utilisateur d'indiquer :

1. la configuration dans laquelle il veut lancer la simulation
2. le nombre de joueurs pour chaque style de jeu

Le plateau ainsi que la liste des joueurs sont créés en fonction des informations indiquées et de paramtres préenregistrés dépendant du mode de simulation choisi. L'État est également ajouté à la liste des joueurs mais est traîté de manière différente des autres joueurs.

2.2 Déroulement du jeu

Le jeu se déroule de la manière suivante : chaque joueur joue à tour de rôle jusqu'à ce que la fin de la partie soit déclarée. Pour chaque joueur, sauf l'État, le dé est lancé, donnant un résultat de 1 à 6. le joueur qui a la main avance d'autant de cases et réalise l'action indiquée dessus. Les actions peuvent être les suivantes :

— Si la case est un investissement et si l'investissement n'est pas encore la propriété d'un joueur, s'il appartient à l'État, le joueur peut décider de l'acheter ou pas, dépendant de son style de jeu et de ses moyens. Si l'investissement appartient déjà à un autre joueur, celui qui joue son tour doit payer au propriétaire un pourcentage de la valeur de l'investissement.

— Si la case est une loi antitrust et si les propriétés du joueur dépassent un seuil fixé par l'État, il doit revendre certaines de ses possessions à moitié prix à l'État pour se trouver

en dessous de ce seuil. Le joueur, dépendant de son mode de jeu choisira les propriétés dont il veut se séparer.
— Si la case est un bureau des finances publiques, le joueur doit payer une taxe sur la somme d'argent qu'il a. La taxe est d'un certain pourcentage indiqué dans la case et est à payer à l'État.
— Si la case est une subvention, le joueur reçoit de l'État le montant indiqué dans la case.
— Si la case est une case de repos, le joueur ne fait rien.

Un joueur sort du jeu s'il n'a plus de moyen financier. Ça se matérialise par un entier passant de 0 (le joueur est encore dans le jeur) à un entier suppérieur, permettant de retrouver l'ordre dans lequel les joueurs sont sortis.
Le jeu prend fin si l'une des conditions suivantes est remplie :

1. Il ne reste plus qu'un seul ou aucun joueur sur le plateau.
2. L'État n'a plus de moyen financier, il a échoué.
3. L'utilisateur décide d'interrompre la partie.

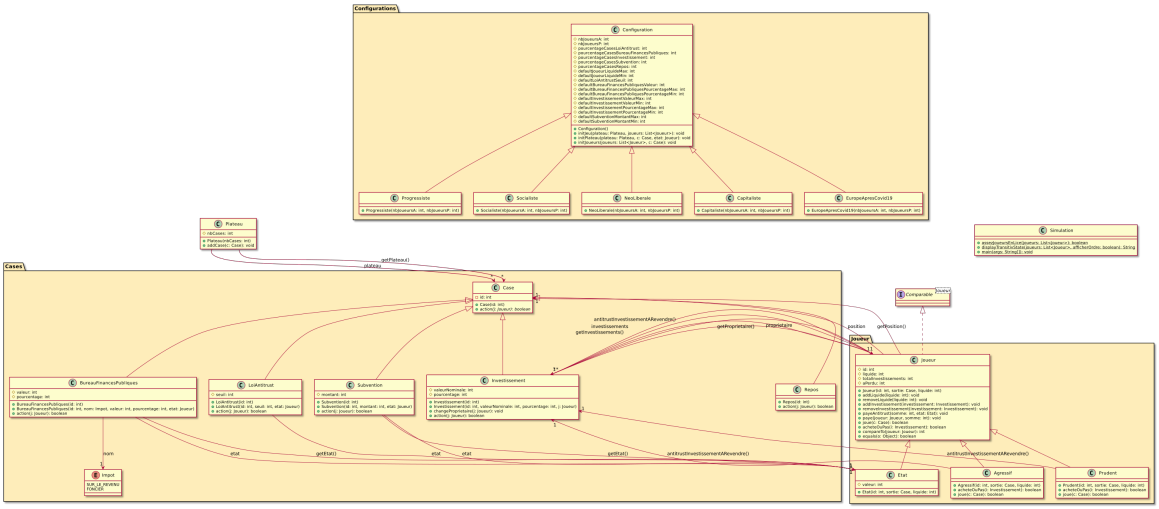
2.3 Résultats

En premier est affiché la raison pour laquelle le jeu a été interrompu.

Les résultats sont affichés par ordre : sont indiqués, le vainqueur, le montant de ses investissements, ses moyens financiers et son patrimoine ; puis vient le tour du second avec les mêmes informations et ainsi de suite.

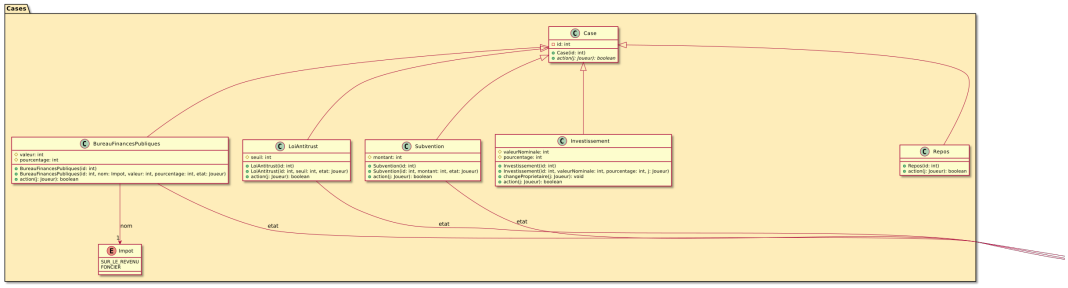
Chapitre 3

Schéma UML des classes

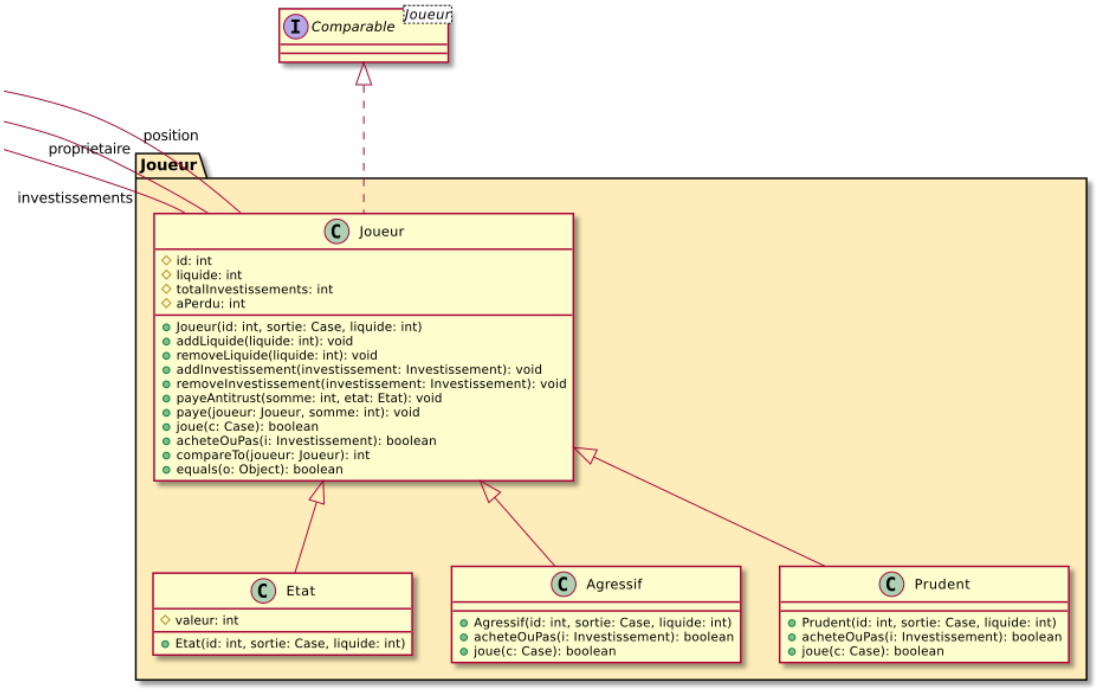


Toute les classes sont regroupées dans un package `Monopoly`.
Le projet est découpé en 5 sous-structures :

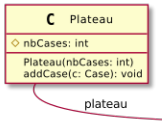
— Le package `Cases` contenant les différents types de cases, chacun représenté par une classe, toutes héritant de la classe `Case`, classe abstraite permettant d'indiquer que tous ces types sont des cases.



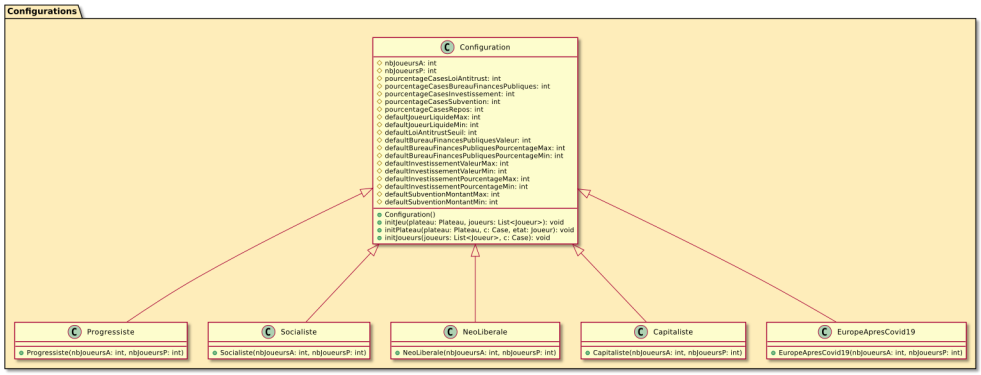
— Le package `Joueurs` contenant les différents styles de jeu, chacun représenté par une classe dont la classe `Etat`, toutes héritant de la classe `Joueur`, classe abstraite permettant d'indiquer que tous ces styles sont des joueurs.



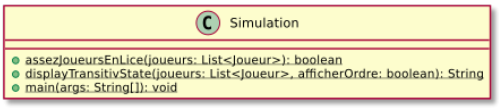
— Le package `Plateaux` contient pour le moment qu'une classe `Plateau` permettant de construire le plateau et d'accéder à ses attributs. Une extension serait possible en ajoutant d'autres plateaux.



— Le package `Configurations` contenant les différents environnements dans lequels le jeu peut se dérouler, chacun représenté par une classe, toutes héritant de la classe `Configuration`, classe abstraite permettant d'indiquer que tous ces types sont des configurations.



— La classe `Simulation`, le main. Contient la création, le déroulement et la fin du jeu.



Chapitre 4

Choix des structures de données non-primitives

La structure non-primitives de la librairie $java.util$ mise en œuvre est la '$LinkedList$'. Elle sert à contenir le plateau, les joueurs ainis que les investissements de chaque joueur.

Le choix de la $LinkedList$ a été fait car les $tableaux$ et $ArrayList$ sont moins efficaces du point de vue de l'ajout et suppression d'éléments.

Chapitre 5

Tests Unitaires

Un test unitaire peut être réalisé par nombre de méthodes dans la classe, autres que constructeur, getteurs, setteurs, égalités, comparaisons, affichage.

Les tests unitaires peuvent être réalisés par package.

La classe Simulation contient une méthode de vérification de la constitution de la liste des joueurs, on peut vérifier que celle-ci renvoit la bonne conclusion.

5.1 Cases

Le package `Cases` : la méthode $action$ peut être testée pour vérifier que l'action liée à la case est effectuée correctement.

5.2 Joueurs

Le package `Joueurs` : les méthode d'achat, $joue$ et $acheteOuPas$ peuvent être testées. Certaines, comme $joue$ nécessiteront plusieurs tests pour s'assurer que l'action a été réalisée correctement.

5.3 Plateaux

Le package `Plateaux` : le plateau ne présente aucune méthode intéressante à tester étant donné qu'il sera construit dans une classe de configuration.

5.4 Configurations

Le package `Configurations` : chaque classe de ce package ne contient aucune méthode sauf la classe `Configuration` construisant aussi bien la liste des joueurs et le plateau, en fonction des demandes de l'utilisateur et des valeurs contenues dans cette classe et celle e la configuration demandée par le joueur. ces méthodes peuvent difficilement être testées étant donnée qu'elles dépendent fortement de l'aléatoire pour que chaque jeu soit différent.

Chapitre 6

Changements réalisés

Les changements opérés entre le premier rapport et l'implémentation sont les suivants :

— Simulation : la simulation n'est plus uniquement constituée du main, mais également d'une méthode d'affichage de l'état de la liste des joueurs et une méthode vérifiant s'il y a encore plus d'un joueur apte à jouer dans la liste des joueurs initiaux (hors État).

— Cases : ajout de la méthode $action$ réalisant l'action du joueur dépendant de son caractère et de la case sur laquelle il est tombé. Ce changement permet de déléguer à chaque case l'action et alléger la fonction joue. Ça permet d'éviter les $instanceof$.

— Joueurs : pour qu'un joueur paye un montant à un autre joueur et éviter de changer les attributs à partir d'une classe extérieur, cette méthode mise en place. Les classes `Agressif` et `Prudent` implémentent également une méthode $acheteOuPas$ pour indiquer si, selon leur caractère, ils achètent l'investissement en paramètre ou pas.

— Plateaux : le plateau n'a pas subi de modifications majeurs à mentionner.

— Configurations : $initPlateau$ et $initJoueurs$ permettent d'alléger la méthode $initJeu$. Ces deux méthodes sont respectivement grandes donc mettre leur code dans une seule méthode la rendrait vraiment très grande, donc difficilement testable, ce qui entraine d'autant plus d'erreurs. Des attributs ont également étés ajoutés, correspondant aux valeurs possibles pour les différentes cases et les montants attribués aux joueurs en lancement de jeu. Les valeurs pouvant varier d'une case à l'autre sont représentées par deux attributs constituant un interval dans lequel les valeurs attribuées aux cases vont se trouver, choisies par de l'aléatoir.

Chapitre 7

Améliorations pouvant être apportées

$initPlateau$ et $initJoueurs$ pourraient être séparés en plusieurs plus petites méthodes pour pouvoir les tester plus aisément.

De plus, il serait intéressant de dispercer ces morceaux dans les classes appropriées : la construction des cases de subvention dans la calsse `Subvention`, les joueurs Prudents dans la classe `Prudent` et ainsi de suite. La méthode $initJeu$ peut rester dans la classe `Configuration` mais déplacer la méthode $initJoueurs$ appelant les nouvelles méthodes de constructions des différents Joueurs dans la classe `Joueur` et chaque méthode de construction de joueur dans sa classe respective. De même, déplacer la méthode $initPlateau$ dans la classe `Plateau` et les méthodes de construction de chaque case dans leurs classes respectives. Cette dernière méthode peut peut-être être une méthode de la super-classe `Case` réalisant la boucle et permettant un $@Override$ dans les classes de chaque case. Ce sera une méthode $abstract$ sinon.

Conclusion

Ce projet nous a permis de mettre en application les notions de Programmation Orientée Objet qui nous ont été enseignées en cours et mettre à l'épreuve nos connaissances fraichement acquises ainsi que la réalisation d'un projet.

Le niveau était très haut et les niveaux le connaissances très différents mais en apportant chacune ses connaissances et son avis, nous sommes arrivées au terme de ce projet avec un programme qui tourne et répond aux attentes.

Nous avons essayé de prendre en compte des contraintes de programmation comme l'espace mémoire et le temps d'exécution pour optimiser notre programme et le rendre le plus efficace et lisible possible et adhérant au mieux au paradigme Orienté Objet.