Oligopoly7

Rapport deuxième itération

Leo Flamencourt, Téo Pisenti, Frederic Camail ,Cylia Yangour, Alexis Courboin, Emile Devos.

Contents

1 Présentation générale du projet

L'objectif de ce projet est d'appliquer les notions de Technologie Objet vues en cours grâce à la mise en oeuvre notre vision du jeu du Monopoly, Oligopol7, qui se joue directement depuis un ordinateur. Il permettra à l'utilisateur de jouer contre d'autres utilisateurs en multijoueurs sur un même écran, au sein d'un terrain 2D isométrique et une interface graphique vivante, surprenante et dynamique. Il mettra notamment en scène la ville de Toulouse et fera références à certains lieux emblématiques du centre ville et de l'ENSEEIHT.

2 Application des méthodes agiles

En suivant les notions vues en méthode agiles, nous avons pris le choix de rendre, pour cette deuxième itération, une version plus poussée de notre Monopoly, avec l'ajout de fonctionnalité et une première version de l'interface graphique. Le but étant de préciser certains aspect du projet, ainsi que d'améliorer les classes et fonctionnalités, en s'approchant le plus possible des "véritables" règles du Monopoly. Ces ajouts se sont organisés sur le git, sous forme d'issues, que nous nous sommes répartis lors qu'une réunion.

Nous avons donc réparti le travail suivant ces nouveaux ajouts ; chaque personne/groupe de personne a géré un aspect particulier (interface graphique, menu, amélioration des fonctionnalités, adaptation/évolution des classes initiales).

3 Explication des fonctionnalités et choix d'implantation

Voici les fonctionnalités implémentées pendant l'itération 1 (avec leur classe associée).

1. Gestion des tours : (arbitre)
    - Lancer le dé (joueur)
    - Avancer le joueur (joueur)
    - Action de la case (caseFonctionnelle)
        - Achat maison si libre et joueur veut (caseLibre)
        - Ne rien faire si veut pas acheter (caseLibre)
        - Payer joueur si maison appartient à autre joueur (caseHotel)
        - Ne rien faire si la maison nous appartient (caseHotel)
    - Fin du tour (arbitre)
2. Changer de joueur automatiquement (arbitre)
3. Arrêter jeu si un joueur n'a plus d'argent (banquerouteException)
4. Afficher les règles (jouer)

5. Joueur la partie (jouer)
6. Afficher les joueurs (plateau)
7. Afficher le plateau (plateau)
    - Afficher une case (caseGraphique)
    - Afficher l'appartenance d'un hotel (caseHotel)
    - Afficher qu'une case est libre (caseLibre)

4 Avancement de l'itération 2

Pour cette 2ème itération, nous avons énuméré différentes "issues" pour améliorer le jeu :

1. Commencer l'implémentation de l'interface graphique
    - Afficher le plateau (partiellement fait, un plateau d'exemple est affiché, il reste à connecter au reste du code)
    - Afficher un pop up graphique : affiche à l'écran un pop up qui propose deux choix : acheter/ne pas acheter. Cela bloque le tour jusqu'à ce que le joueur clique sur un des boutons et fasse son choix. (à faire)
    - Afficher un menu principal : un menu est montré au lancement de l'application. Il permet de lancer une partie, reprendre une partie en cours, voir les crédits. (à faire)
    - Afficher un menu jeu : création des boutons pour lancer le dé, ouvrir le menu des propriétés, finir le tour courant. Afficher le scoreboard en haut à droite (avec la couleur du joueur qui joue) (partiellement fait, les boutons sont cliquables mais affichent juste un texte dans la console quand on clique dessus pour le moment.)
2. Améliorer les fonctionnalités liées aux propriétés
    - Implémenter un code pour l'enchère d'une propriété : si un joueur refuse d'acheter une propriété, elle est mise aux enchères. N'importe quel autre joueur peut alors enchérir les uns après les autres et celui qui met le tarif le plus élevé gagne la propriété.
    - Améliorer les propriétés : une propriété peut être améliorée à tout moment depuis le menu des propriétés grâce à son nom (ajouts d'hôtels, etc)
    - Hypothéquer une propriété : une propriété peut être hypothéquée à tout moment (pendant son tour) depuis le menu des propriétés. Le joueur récupère alors la somme de l'hypothèque mais ne gagne plus d'argent si d'autres joueurs tombent sur sa propriété hypothéquée. Il peut racheter cette hypothèque pour 10% de plus depuis le même menu pour revenir à l'état antérieur.

3. Améliorer les fonctionnalités liées aux cartes et case prison

   - Carte chance : ajout des cases "cartes chances" qui déclenchent des événements que l'on peut customiser. Une carte chance a un texte qui explique ce qu'elle fait et une fonction d'événement associée.
   - Case prison : ajout d'une case prison où le joueur est enfermé jusqu'à qu'il fasse un double ou paie de l'argent.
   - Aller en prison : ajout d'une case qui envoie un joueur en prison. Celui-ci ne passe pas par la case départ. Ajout d'une carte chance qui envoie en prison et une carte chance "sortir de prison" qui est la seule carte que le joueur peut garder et le rend "immunisé" à la prison.

4. Adapter la classe arbitre : ajout/adaptation des actions déclenchés par des boutons cliqués par le joueur.

5 Diagramme de classe (UML)

Ci-dessous notre diagramme de classe UML (premier "brouillon" puis UML final) :

Justification / explication : L'interface CaseFonctionnelle et CaseGraphique permettent de factoriser/généraliser le code des cases (hôtel, libre). Elles sont réalisées par les classes CaseHotel et CaseLibre qui redéfinissent les méthodes executer() (interface CaseFonctionnelle) et afficher() (interface CaseGraphique). On ajoute les attributs publiques propriétaire, valeur et position à CaseHotel pour y avoir accès depuis les autres classes du jeu et faciliter les échanges d'informations entre elles. Idem pour CaseLibre, avec les attributs valeur et position.

Ces interfaces sont reliées par des relations d'agregagtion/composition avec la classe Plateau, puisqu'un plateau n'a pas de sens sans case.

Pour la dynamique du jeu, nous avons fait le choix de créer une classe Arbitre avec une méthode publique arbitrer() qu'on appelle dans la classe Plateau par exemple. Cette classe est reliée à une classe Jouer par une relation de composition (l'arbitre n'existe pas sans jeu, et la fin du jeu signe la fin de vie de l'arbitre également), une classe qui sert notamment à lancer le Jeu.

Nous avons enfin fait le choix de créer une classe Joueur pour gérer les différents objets "joueurs" représentés par leur nom, avatar et solde. Plusieurs méthodes publiques permettent de gérer le jeu des joueurs comme créditer() ou getSolde() par exemple. Cette classe est reliée à la classe Plateau par une relation d'agrégation (un plateau a un/des joueurs).

6 Explication de l'architecture

- le package par défaut (à la racine du dossier src) contient une classe "HelloBis" qui sert à lancer l'interface graphique (qui n'est pas encore

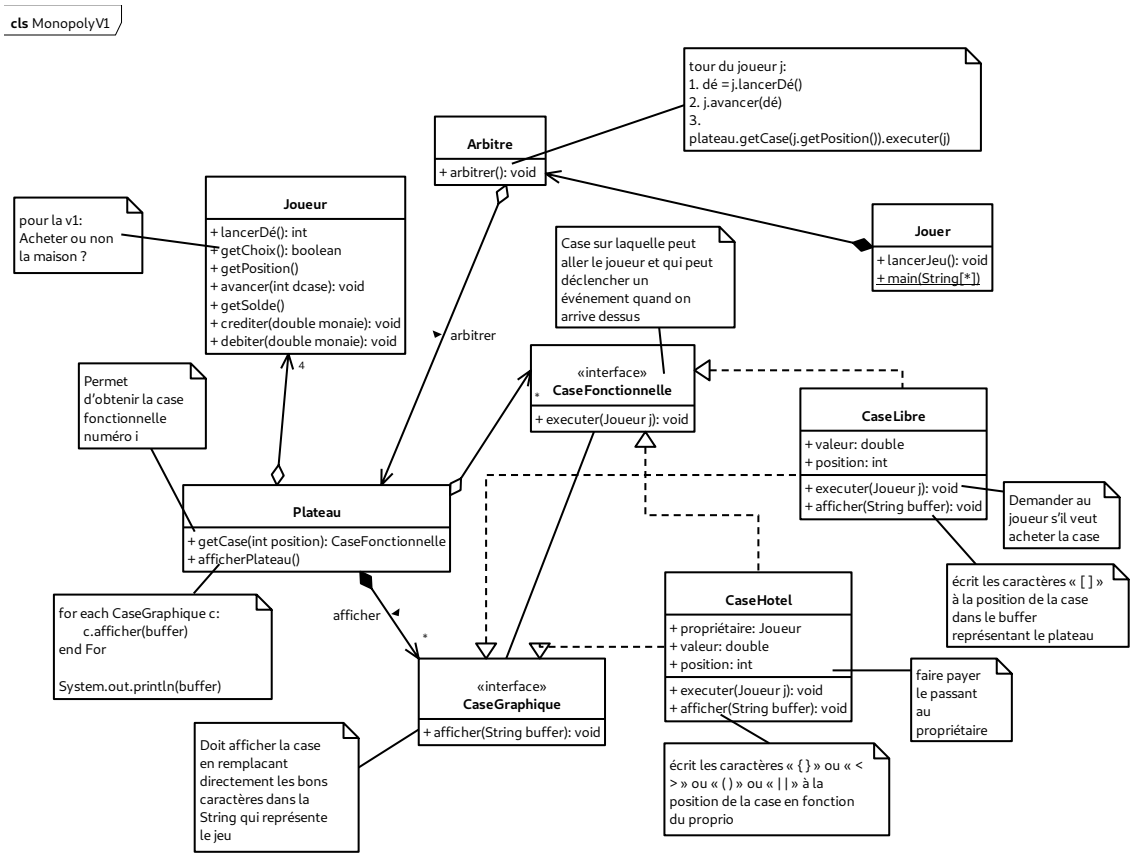

Figure 1: Diagramme de classe de la logique du Monopoly

intéractive) seulement et une classe "Jouer" qui sert à lancer le vrai jeu avec une interface en ligne de commande.

- le package "logiqueMonopoly" contient toutes les fonctions qui permettent de gérer une partie de Monopoly (gestion des tours, des joueurs, de l'argent, des propriétés, des cartes chances et prison...). Il permet de faire fonctionner le jeu dans sa version non graphique.

- le package "interfaceGraphique" contient les éléments de l'interface graphique (boutons, plateau, popup graphiques...) qui seront affichés à l'écran au cours de la partie.

- le package "moteurGraphique" est un moteur graphique avec Opengl (rendu accéléré par la carte graphique) qui permet d'afficher des objets à l'écran. Il est utilisé par les composants du package "interfaceGraphique".

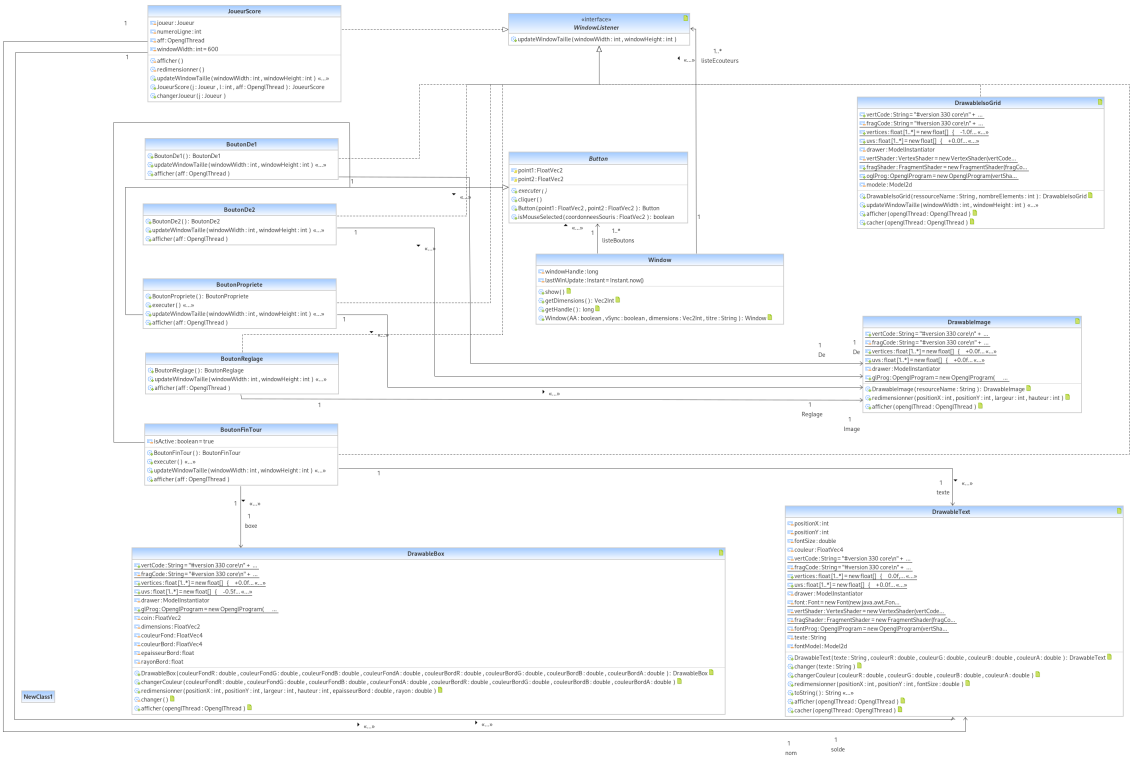

Figure 2: Diagramme de classe de l'interface graphique

6.1 Présentation des fonctionnalités du moteur graphique

Le moteur graphique utilisé met à disposition du programmeur les classes suivantes (voir l'UML qui suit). Celles-ci sont décrites dans les sous-packages "moteurGraphique.drawable" et "moteurGraphique.window". Les autres classes permettent au moteur de fonctionner mais ne sont pas utiles pour le programmeur.

(Remarque: vous pouvez retrouver tous les diagrammes de classe dans le dossier doc/architectures sur le git)

Sur ce diagramme, on reconnait à gauche les classes des principaux composants de l'interface graphique du Monopoly (package "interfaceGraphique") :

- JoueurScore le "scoreboard" avec le classement des joueurs
- BoutonDe1 et BoutonDe2 les boutons qui permettent de lancer les dés
- BoutonPropriete et BoutonReglage les futurs boutons pour ouvrir le menu des propriétés ou des paramètres
- BoutonFinTour le bouton qui permet de finir son tour

Ces objets sont définis à partir des classes du package "drawable" du moteur graphique :

- DrawableBox permet de créer un rectangle avec une bordure et des coins arrondis dont on peut choisir les couleurs.
- DrawableImage permet d'afficher une image du dossier "res/" à l'écran.
- DrawableText permet d'afficher une chaine de caractère à l'écran dont on peut choisir la couleur.
- DrawableIsoGrid permet de créer une grille isométrique. Celle-ci est définie par une image qui contient tous les éléments possibles dans cet espace isométrique avec son propre système de coordonnées. On peut changer le zoom lors de l'affichage de la grille à l'écran et déplacer une caméra dans l'espace isométrique pour changer l'objet affiché au centre. Les objets qui débordent de la grille sont automatiquement masqués.

Les classes des objets du package "moteurGraphique.drawable" contiennent les méthodes suivantes (vous pouvez lire la javadoc pour en savoir plus)

- afficher(OpenglThread) qui permet d'afficher l'objet à l'écran. Le paramètre est une poignée vers l'objet chargé de la boucle d'affichage.
- redimensionner(...) permet de replacer l'objet graphique à l'écran à partir de la coordonnée du point suppérieur gauche, et de la taille de l'élément graphique. Ces valeurs sont en pixels, l'origine du repère étant le coin suppérieur gauche de la fenêtre.

Les objets ont souvent besoin de connaître la taille actuelle de la fenêtre (par exemple pour redimensionner les objets affichés à l'écran). Pour celà, il suffit qu'ils réalisent l'interface "WindowListener", et qu'ils définissent la méthode "updateWindowTaille". Elle est appelée à chaque redimenssionement de la fenêtre courante et passe la nouvelle taille de fenêtre (en pixels) en paramètre.

Enfin, on peut créer un bouton en héritant de la classe "Button" du package "moteurGraphique.window". Il faut alors affecter à point1 et point2 les coordonnées des coins suppérieur gauche et inférieur droit de la boite de colision du bouton (zone dans laquelle on peut cliquer). Et il faudra redéfinir la méthode abstraite "executer()" par le handler de notre bouton.