Rapport projet de POOIG

Azul

Binôme : SOUFFAN Nathan et BOUARAH Romain

Réalisation

Notre projet consiste en une implémentation en Java du jeu de société d'Azul. C'est un jeu de société en tour par tour. Notre projet permet de jouer en mode console. Enfin, le jeu peut se jouer de 2 à 4 joueurs sur un même ordinateur.

Partie logique

Structure générale

Les zones de jeu

Chaque zone de jeu possède une classe la représentant. Ces classes agissent comme un intermédiaire entre l'utilisateur et les listes de tuiles, cela permet de mieux contrôler l'interaction possible. Cela ajoute certes de la compléxité mais grâce à ces classes le modèle reste cohérent.

Le joueur

Le joueur est représenté par la classe abstraite *Player*. Elle est hérité par la classe *HumanPlayer*. Nous avons décider de choisir cette représentation pour garder la possibilité d'ajouter des joueurs non humain.

La classe *Player* possède entre autre une classe interne *PlayerBoard.* Nous avons utilisé la classe *PlayerBoard* pour regrouper les classes : *Floor, PatternArea* et *Wall*, parce que chaque joueur possède ces trois zones qui interagissent entre elles. De plus, *PlayerBoard* est interne car elle n'a de sens que pour un joueur.

Enfin, la classe a une méthode abstraite *play* prenant un argument un *InputManager.* Le joueur doit en effet pouvoir interagir avec la partie lorsqu'il en est autorisé.

Le jeu

La classe *Game* est le cœur logique du jeu. Elle possède donc une liste de tous les joueurs. Elle a aussi une classe interne *GameBoard* pour représenter le plateau de jeu avec ses fabriques, son centre, le sac et la défausse.

Azul est divisé en trois phases : une phase de préparation, d'offre et de décoration. Ces phases sont représentées par des méthodes dans *Game*. Par conséquent, la boucle de jeu ne se situe pas dans *Game* mais dans *GameController* qui permet de faire le lien entre la vue et le modèle.

Les tuiles

Il existe une classe *Tile* dont dérivent les classes *ColoredTile* et *FirstTile*. Pour représenter les couleurs nous avons eu recours à un type enuméré interne à *ColoredTile*. Enfin, il existe une interface *SpecialTile* ayant pour seul méthode *effect* qui prend un argument *Game*, en effet chaque tuile spécial applique son effet de manière globale au jeu.

Au début du projet, nous ne savions pas comment implémentater les tuiles joker, c'est pour cela qu'on a gardé cette strucutre, pour nous réserver une possibilité.

Variantes du jeu

Nous n'avons pas implémenté de variantes de jeu par manque de temps et d'organisation. En revanche, nous avons réfléchie à comment les implémenter. Nous expliquons cela en détail dans Problèmes rencontrés.

Partie Affichage

Le jeu peut être lancé en mode graphique (non fonctionnel) ou en mode console, cela est possible car nous avons bien séparé la vue du modèle.

La classe abstraite *GameRenderEngine* représente le moteur graphique du jeu ; c'est la vue. Nous avons fait le choix de ne pas optimiser l'affichage pour plus de simplicité, le moteur doit dans son comportement redessiner entièrement la vue à partir du modèle et non qu'une partie.

La classe abstraite *InputManager* est utilisé par la méthode *play* dans *Player* pour permettre au joueur d'effectuer ses actions sur la partie. Cet *InputManager* est hérité en fonction du besoin du moteur de jeu. Par exemple, *ConsoleInputManager* permet de communiquer avec le jeu via les entrées consoles.

La boucle de jeu

D'après le sujet la boucle de jeu devait se trouvait dans *Game* , mais nous trouvions cela étrange d'avoir des appels à la vue dans le modèle.

C’est pourquoi, nous introduisons la classe *GameController* dans laquelle se trouve la boucle de jeu. La boucle de jeu doit pouvoir mettre à jour le modèle, lire les entrées utilisateurs et mettre à jour la vue. C’est pour cela qu’elle se trouve dans cette classe et non dans *Game* car *GameController* doit faire le lien avec *Game* et un *GameRenderEngine*.

Problèmes rencontrés

Organisation

Nous avons passé plus de quatre heures à planifier la structure du projet mais cela n’a pas suffit nous avons donc eu des problèmes d’architecture durant la période de programmation et cela nous a beaucoup retardé, donc notre version jouable est arrivé très tard. Nous pensons qu’avec plus de temps nous aurions sans doute répondu en entier au sujet.

Les variantes de jeu

Les deux variantes ont en commun une interaction possible durant la phase de décoration.

Pour réaliser cela, on crée une interface *DecorationPhase* possédant une méthode *process(InputManager im)* et une méthode *process()*. Cette interface est ensuite implémentée par des classes *NormalDecorationPhase*, *JokerDecorationPhase*, ... Ces classes n’implémenteront que les méthodes possibles et dans la cas échéant renverront une exception.

La classe *Game* devra alors juste choisir la bonne classe. Ce choix peut être fait à la création d’une instance de *Game* à l’aide du design pattern *Factory*.

Tuile joker

Même si nous n’avons pas implémenté cette variante nous avons quand même réflechi à son implémentation. On rajoute une couleur *JOKER* dans le type enuméré des couleurs. Avec les modifications d’au-dessus et quelques autres modifications la variante peut être jouée.

Mur sans couleur

La variante du mur sans couleur nous semble plus difficile. Comme pour *DecorationPhase*, il faut créer une interface *Wall* possédant une méthode *add(int column)* et une méthode *add()*.

De même, les classes choisissent au moins une méthode à implémenter.

Avec cette variante, le joueur peut se retrouver bloqué. Pour empêcher cela, on autorise le joueur à poser des tuiles de même couleur sur une même colonne en échange d’un malus de points. On garde évidemment le fait de ne poser qu’une seule couleur par ligne.

L’interface graphique

Le mode graphique de notre projet n’est pas fonctionnel, le modèle n’est pas pris en compte dans sa totalité, en effet nous avons décidé d’abandonner le mode graphique par manque de temps.

La principale difficulté était le *Drag and Drop* des tuiles. Nous représentons les tuiles par des *JLabel.* Pour permettre le *Drag and Drop* sur des *Jlabel* , il faut, d’après la documentation java, appeler la méthode *setTransferHandler(TransferHandler th)*. Dans les premières versions du jeu, le *Drag and Drop* fonctionnait du côté de la vue. Le problème est que le jeu doit se mettre en pause le temps que le joueur fasse une action, malheuresement nous avons pas réussi à intégrer le *D&D* dans la boucle de jeu et avons décidé de complétement retirer le *Drag and Drop*.

Pistes d’extensions

Interaction avec le jeu

Pour mieux mettre en évidence le rôle joueur de la classe *Player*. La méthode *play* doit renvoyer une *GameAction* et c’est la classe *Game* qui applique cette *GameAction*. Avec cette séparation on voit clairement qu’un *Player* n’émet que des actions et *Game* ne reçoit que des actions. Ainsi, on voit clairement qu’il y a un ensemble fini d’actions.

Joueur non humain

Nous avons laissé la possibilité d’intégrer des joueurs non humain. Il faut hériter de la classe *Player*. Le jeu devra être représenter par un état initial, un ensemble de joueurs, un ensemble d’actions possibles pour un état donné, un modèle de transition, un test de terminaison et une fonction d’utilité. Avec cette représentation, l’I.A. pourra générer un arbre d’actions.