Rapport du projet d'Hackenbush

GUYOT Yoann
LOPES Guillaume

23 Avril 2007

Table des matières

Le but du projet est d'implémenter le jeu Hackenbush. Ce jeu comporte deux parties : une partie édition et une partie jeu. Nous avons implémenté ce projet en restant fidèle au modèle MVC. Dans la suite, nous décrivons les classes de chaque catégorie du modèle.

1 Les classes du Modèle

Dans cette partie, nous décrivons toutes les classes qui composent le modèle du projet.
Tout d'abord, nous devons distinguer que dans ce projet, il y a deux types de ligne. Il y a le sol (la ligne noire), qui ne peut être ni déplacé, ni supprimé et ni modifié, et aussi les segments (les lignes rouges, bleues ou vertes) qui peuvent être déplacés, supprimés ou modifiés.

Ligne :
C'est une classe abstraite, qui représente toutes les lignes du jeu, c'est à dire le *Sol* et les *Segment*.
Elle contient quatre coordonnées, une couleur et un numéro (cf. la classe GrapheLigne).
De plus, elle ne possède qu'une seule fonction abstraite :
tracer(Graphics2D gBuffer).
Cette fonction permet de dessiner la ligne dans le "buffer" graphique donné.

Sol :
Cette classe étend *Ligne*. Elle représente le sol (la ligne noire) du jeu.
Elle se contente de définir la fonction *tracer(Graphics2D gBuffer)*.

Segment :
Cette classe étend *Ligne*. Elle représente les segments du jeu (les lignes rouges, bleues ou vertes).

GrapheLigne :
Cette classe représente toutes les lignes d'un plateau d'édition ou de jeu. Nous avons choisis dans notre implémentation de représenter toutes les lignes d'un plateau de jeu ou d'édition dans un graphe non-orienté. Dans ce graphe, les sommets sont les lignes et les transitions sont données par le fait que les extrémités d'une ligne coincide avec une autre. Les champs de la classe GrapheLigne sont :
- *int nbLignes* le nombre de lignes du graphe
- *ArrayList<Ligne> lignes* la liste de lignes du graphe
- *ArrayList<ArrayList<Ligne» transitions* la liste d'adjacence du graphe
- textit Sol sol le sol, la ligne à laquelle les autres lignes doivent connectés (directement ou non) sert pour le mode Jeu

Pour le plateau d'édition, on rajoute des lignes au graphe sans leur créer de transition, en effet, cela n'a aucun sens de créer des transitions pendant la création du plateau ! Les transitions ne sont créés qu' au moment de la sauvegarde du plateau.
Pour le plateau de jeu, on a un "vrai" graphe auquel on lance un parcours en largeur à chaque fois que l'utilisateur retire un segment du plateau.
Player :
Cette classe représente un joueur humain.Elle possède le nom du joueur et sa couleur.

Game :
Cette classe représente une partie. Elle possède quatre champs :
- *GrapheLigne grapheLignes* le graphe contenant toutes les lignes du jeu
- *Player rouge* le joueur rouge
- *Player bleu* le joueur bleu
- *Player currentPlayer* le joueur courant

Toutes les classes du modèle sont Serializable.

2 Les classes de la Vue

Dans cette partie, nous décrivons toutes les classe de la vue.

Dans notre projet, nous avons une *Fenetre* principale, qui contient un *Menu*. A partir de la, l'utilisateur peut décide lancer un jeu (si un jeu a été créé auparavant) ou alors de lancer l'éditeur de jeu. Il faut noter cependant, qu'une fois que l'utilisateur lance un jeu ou l'éditeur, le *Menu* est toujours visible. Cela permet à l'utilisateur de passer du jeu à l'édition simplement !

Fenetre :
Cette classe étend la classe *JFrame*, c'est le conteneur principal de notre application, c'est lui qui va contenir tous les autres conteneurs. Regardons les champs de cette classe :
- *Menu menu* la barre de menu
- *Plateau plateau* la plateau de jeu (seuleument pour le mode jeu)
- *StatusBar statusBar* la barre de statut du jeu qui indique quel joueur doit jouer et permet de défaire/refaire un coup
- *TabbedPlateau tabs* l'ensemble d'onglets contenant les plateaux en cours d'édition
- *Tools tools* la barre d'outils (en mode édition)

Nous pouvons regrouper en deux parties les champs de la classe *Fenetre*(sauf Menu), ceux pour l'édition (TabbedPlateau et Tools) et ceux pour le jeu (Pla-

teau et StatusBar).
Avant cela, nous aimerions présenter la classe *Menu*.
Menu : Cette classe étend la classe *JMenuBar*. Cette classe contient toutes les options que l'utilisateur peut réaliser. En effet, quand l'utilisateur lance l'application, il se trouve devant deux menus : Jeu et Editeur.
Dans le menu Jeu, il peut commencer une Nouvelle partie (si il en a déja créé une), Sauvegarder une partie (s il est en train de jouer), Charger une partie (qui a été sauvegardé préalablement) et Quitter le jeu. Dans le menu Editeur, il peut créer un Nouveau plateau, Sauvegarder un plateau et Charger un plateau.
Des raccourcis claviers ont été posés sur toutes ces options.

2.1 Les classes du mode Edition

Dans la partie éditeur de notre application, nous avons une barre d'outils (*Tools*) et un ensemble d'onglets contenant les plateaux en cours d'édition. Ainsi l'utilisateur peut créer plusieurs plateaux en même temps !

Tools :
Cette classe étend la classe *JToolBar*. Elle contient quatres boutons :
– *JButton undo* permet d'annuler une action
– *JButton redo* permet de rétablir une action
– *creerSegment* permet de créer un segment
– *deplacerSegment* permet de déplacer un segment
Elle contient aussi trois *JRadioButton* qui permettent de sélectionner la couleur du segment que l'on veut créer.
Il faut noter que le fait de pouvoir créer un segment sur un plateau implique que l'on peut créer aussi un segment sur tous les plateaux. Par exemple, l'utilisateur clique sur le bouton de création d'un segment (celui-ci reste enfoncé) , même s'il change de plateau, il pourra toujours créer un segment.De même, pour déplacer un segment.
D'ailleurs, nous avons un problème avec les undo/redo dans l'éditeur. Nous espérons que d'ici la soutenance ce problème sera résolu.

TabbedPlateau :
Cetta classe étend *JTabbedPane*. Elle contient tous les *Plateau* en cours de création. Les plateaux sont présentés sous forme d'onglets, ainsi l'utilisateur peut avoir plusieurs plateaux sous les yeux.

2.2 Les classes du mode Jeu

Dans la partie jeu de notre application, nous avons une barre de statut (*StatusBar*) et un plateau (*Plateau*).

StatusBar :
Cette classe étend la classe *JPanel*. Les champs de cette classe sont :
– *JButton undo* permet d'annuler le coup d'un joueur
– *JButton redo* permet de rétablir le coup d'un joueur
– *JLabel labelTurn* indique quel joueur doit jouer ou quel joueur a perdu
Cette barre de statut permet de savoir quel joueur doit jouer ou a perdu et de défaire/rétablir les coups d'un joueur.

Plateau :
Cette classe étend la classe *JPanel*. C'est dans ce panel que l'on va dessiner les segments.
Elle contient un *GrapheLigne*, qui sont donc les lignes du jeu. Dans la classe *Plateau*, nous avons redéfini la fonction : *paintComponent(Graphics g)*. Cette fonction se contente de dessiner tous les segments du *GrapheLigne*.

3 Les classes du Controleur

Dans cette partie , nous décrivons les classes du Controleur.

Comme pour la section précédente, nous pouvons regrouper les écouteurs en plusieurs groupes.
Nous avons d'abord l'écouteur principal *MenuListener* posé sur le *Menu*. Ensuite, les écouteurs pour l'édition de plateau (*EditionListener, MovingModeListener et CreationModeListener*) et les écouteurs pour le jeu (*GameListener*). Enfin, nous avons les écouteurs pour les undo/redo (*GameUndo et EditionUndo*).

MenuListener :
Cette classe implémente l'interface *ActionListener*. Elle contient deux champs importants :
– *Fenetre fenetre* la fenetre principale de l'application
– *Game game* une partie de Hackenbush
C'est écouteur permet d'exécuter les actions vus dans la section précédente.

3.1 Les écouteurs de l'Editeur

Pour la partie éditeur, nous avons plus d'écouteur que la partie jeu. En effet, les possibilités de l'utilisateur sont plus grandes.

EditionListener :
Cette classe implémente l'interface *ActionListener*. Cet écouteur est posé sur les boutons de l'objet *Tools*. Elle contient trois champs importants :
- *Fenetre fenetre* la fenetre principale de l'application
- *CreationModeListener cMode* l'écouteur pour le mode création/modification de segment
- *MovingModeListener mMode* l'écouteur pour le mode déplacement/suppression de segment

C'est cet écouteur qui pose le "bon" écouteur sur tous les plateaux. En effet, cet écouteur s'occupe de mettre l'écouteur en fonction du bouton sur lequel l'utilisateur à cliqué.
Par exemple, le fait de cliquer sur le bouton de création de segment de la barre d'outils (*Tools*) supprime l'écouteur *MovingModeListener* et met à la place l'écouteur *CreationModeListener* sur le plateau. De même, si on clique sur le bouton de déplacement d'un segment.

CreationModeListener :
cette classe implémente l'interface *MouseInputListener*. Cet écouteur permet la création ou la modification d'un segment sur le plateau sélectionné. Pour créer un segment, l'utilisateur doit rester appuyer sur le bouton gauche de la souris. A l'endroit ou il a commencé a appuyer se trouvera le point de départ du segment et la ou il relachera se trouvera le point d'arrivé. Pour modifier la taille d'un segment, l'utilisateur doit positionner le curseur de la souris sur une extrémité d'un segment et rester appuyé sur le bouton droit de la souris. E n déplaçant la souris, il modifie ainsi sa taille. Une fois le bouton de la souris relaché, la taille du segment est fixé.

MovingModeListener :
cette classe implémente l'interface *MouseInputListener*. Cet écouteur permet le déplacement ou la suppression d'un segment sur le plateau sélectionné.
Pour déplacer un segment, l'utilisateur positionne le curseur de la souris sur le segment et reste appuyer sur le bouton gauche de la souris.En déplaçant la souris, il déplace le segment. Une fois le bouton relaché, le segment reste à sa position.
Pour supprimer le segment, l'utilisateur doit faire un clik droit sur le segment qu'il veut supprimé.

Les deux écouteurs précédents sont posés sur tous les plateaux en cours de création !

De plus, notre application permet facile de mettre "bout à bout" des segments. En effet, quand l'utilisateur "approche" l'extrémité d'un segment à une autre extrémité, les deux extrémités se superposent automatiquement.
EditionUndo :
Cette classe étend la classe *AbstractUndoableEdit*. Elle représente une action annulable dans le mode éditeur.
Pour réaliser les undo/redo, nous gardons une liste des lignes avant l'action et une autre liste de l'action aprèes l'action. Enfin, nous redéfinissons les méthodes : *undo() et redo()*.

3.2 L'écouteur du Jeu

La partie du jeu ne comporte qu'un seul écouteur.

GameListener :
Cette classe implémente les interfaces *ActionListener, MouseInputListener*. Cet écouteur est posé sur le *Plateau* du jeu et sur les boutons de la *StatusBar*.
Cet écouteur ne gère que le clik sur le plateau. Si l'utilisateur clique sur un segment cela supprime le segment (et aussi les segments qui ne sont plus connectés).
Pour les boutons undo/redo de la *StausBar*, il s'occupe d'effectuer les opérations d'annulation et de rétablissement d'un coup.

GameUndo :
Cette classe étend la classe *AbstractUndoableEdit*. Elle représente une action annulable dans le mode jeu.
Pour réaliser les undo/redo, on garde une liste des segments qui ont été retirés en jouant un coup. Enfin, nous redéfinissons les méthodes : *undo() et redo()*.

3.3 Conclusion

L'application que nous vous donnons correspond au programme minimal sans erreur.Nous avons implémenté une extension dans la partie éditeur qui permet à l'utilisateur d'effectuer des undo/redo. Cette dernière extenstion ne fonctionne pas très bien dans cette version, mais nous espérons corriger ce bug pour la soutenance.