Le labyrinthe, partie I.

POnGL, TP4, modèles de conception.

Introduction

Dans ce TP et le prochain, nous allons réaliser un petit jeu, dans lequel un personnage doit s'échapper d'un labyrinthe sans se faire attraper par ses gardiens. Ce double TP sera un prétexte à mettre en pratique certains des modèles de conception décrits au dernier cours, à en voir quelques nouveaux, et à réemployer tout ce que nous avons vu pendant le semestre.

Aux adresses ci-dessous, vous trouverez un squelette à compléter et à enrichir progressivement au gré des sections de cette feuille, ainsi qu'un fichier de démonstration illustrant ce que pourra faire votre programme si vous implémentez toutes les classes et méthodes décrites ici :
`http://www.lri.fr/~blsk/POnGL/Laby.java`
`http://www.lri.fr/~blsk/POnGL/Laby.jar`

L'architecture MVC

Pour organiser notre programme et son interface graphique, nous utiliserons le modèle de conception architectural Modèle-Vue-Contrôleur (MVC), dans lequel trois rôles sont distingués :

1. Le *modèle* est le cœur du programme, qui contiendra le labyrinthe proprement dit, ainsi que toutes les méthodes régissant ce qui se passe dans ce labyrinthe.
2. La *vue* est l'interface graphique du programme, qui va afficher le labyrinthe (et qui, la semaine prochaine, contiendra également des boutons).
3. Le *contrôleur* récupère les actions du joueur (touches du clavier pressées, boutons cliqués) et déclenche les actions du jeu correspondantes.

La structure du programme

Le squelette contient quatre classes principales :

1. La classe `Laby` est la classe principale, qui met en place l'architecture MVC. Elle initialise le modèle, la vue, le contrôleur, et ouvre une fenêtre graphique. Vous n'avez pas à y toucher dans un premier temps.
2. La classe `LModel` décrit le labyrinthe proprement dit. Pour l'instant, elle contient simplement dans des commentaires des méthodes qui permettent d'initialiser le labyrinthe (vous les "décommenterez" progressivement, quand votre programme aura les attributs/méthodes/classes requises). Vous devrez compléter cette classe.
3. La classe `LView` décrit un composant graphique dont on voudra qu'il affiche le contenu du labyrinthe. Vous devrez compléter cette classe.
4. La classe `LController` décrit un contrôleur qui répond aux entrées clavier, et qui les traduit en instructions pour le modèle.

Différentes classes seront à ajouter progressivement à cette épine dorsale, notamment pour représenter les cases du labyrinthe et les différentes créatures pouvant y évoluer.

1 Construire et afficher un labyrinthe

Dans le modèle, on représentera le labyrinthe par un tableau de cases. Les cases seront de la classe `Cell`. On veut également des classes `Wall` et `Exit`, respectivement pour les cases qui sont des murs et les cases qui sont une sortie du labyrinthe. On veut que les cases normales s'affichent en blanc, que les murs s'affichent en noir, et que les sorties s'affichent en bleu. Une classe `Cell` est déjà proposée avec une méthode gérant l'affichage en blanc. Le travail à effectuer est le suivant :

1. Écrire le constructeur `LModel(int h, int l)` de la classe `LModel`, où les arguments `h` et `l` représentent la hauteur et la largeur du labyrinthe (en nombre de cases). Ce constructeur devra initialiser le tableau avec des cases normales.
2. Repérer dans la méthode `Cell:paintCell` l'endroit où la couleur est définie. Modifier cette méthode pour lui permettre d'être réutilisée par héritage par des cases s'affichant avec d'autres couleurs que le blanc.
3. Définir les classes `Wall` et `Exit` comme sous-classes de `Cell`, de sorte qu'il ne soit pas nécessaire de redéfinir `paintCell`. Vous pourrez avoir à introduire d'autres méthodes en revanche.
4. Dans la classe `LView`, compléter la méthode `paintComponent` pour appeler les méthodes `paintCell` de toutes les cases du labyrinthe.

Pour tester cette partie, décommentez les méthodes `putWall` et `putExit` dans la classe `LModel`, et la première section de configuration dans la classe `Laby`.

2 Introduire des créatures dans le labyrinthe

On veut que chaque case puisse contenir (au plus) une créature. On aura deux classes particulières de créatures : `Hero` et `Monster`. On veut que toutes les créatures possèdent une méthode `paintCreature`, qui affiche un rond bleu pour les héros et gris pour les monstres. Le travail à effectuer est le suivant :

1. Définir les classes `Hero` et `Monster`, plus d'éventuelles autres classes vous aidant à les hiérarchiser.
2. Ajouter à la classe `Cell` les attributs et méthodes permettant d'accueillir un occupant éventuel (on peut utiliser l'objet nul `null` pour les cases inocuppées).
3. Définir pour vos créatures une méthode de signature `public void paintCreature(Graphics2D g2d, int leftX, int topY, int scale)`, bâtie sur le modèle de la méthode `paintCell`. Vous pourrez utilisez la classe `Ellipse2D` à la place de `Rectangle2D`.
4. Faire en sorte que l'affichage d'une case affiche aussi son éventuel occupant.

Pour tester cette partie, décommentez la méthode `putHero` dans la classe `LModel`, et la deuxième section de configuration dans la classe `Laby`.

3 Animer le héros

On veut maintenant que le labyrinthe contienne un héros qu'on peut déplacer d'une case vers le haut, le bas, la droite ou la gauche en pressant les touches `e`, `c`, `f` ou `s`. Les directions sont représentées par la classe d'énumération introduite par `enum Direction` dans notre fichier. Cette énumération propose les quatre directions `NORTH`, `SOUTH`, `EAST`, et `WEST`, chacune associée à une paire d'entiers qui correspond à une différence de coordonnées dans le tableau. Ces différences de coordonnées peuvent être récupérées avec les méthodes `dI()` et `dJ()`.

À cette étape, la séparation des tâches entre le modèle, la vue et le contrôleur et les relations entre ces trois classes va devenir apparente.

— Le contrôleur possède une méthode `void KeyTyped(KeyEvent e)` qui intercepte une frappe au clavier, et analyse la touche qui a été utilisée. Si cette touche correspond à une direction valide, le contrôleur appelle la méthode `LModel:heroMove(Direction dir)`.
— Le modèle possède une méthode `void heroMove(Direction dir)` qui déplace le héros dans le labyrinthe, d'une case dans la direction `dir` fournie en argument. Si un mouvement a eu lieu, les méthodes du modèle doivent appeler successivement les méthodes `setChanged()` et `notifyObservers()` pour indiquer à la vue qu'elle doit se remettre à ajour.
— La vue possède une méthode `update`, qui déclenche le rafraîchissement de l'affichage quand elle reçoit un signal indiquant que le modèle a été modifié.

La relation entre le contrôleur et le modèle est directe : le contrôleur connaît le modèle, et appelle directement ses méthodes (comme `heroMove`). La relation entre le modèle et la vue en revanche n'est pas directe ; elle se fait par l'intermédiaire d'un modèle de conception "Observateur" (*Observer*). Dans ce modèle on a deux rôles :

— Un objet observable, qui peut évoluer avec le temps, et qui envoie un signal quand une évolution a lieu. C'est ici le modèle (notez le `extends Observable` dans la définition de la classe `LModel`).
— Un objet observateur, qui se met à jour quand un signal est lancé par l'objet observé. C'est ici la vue (notez le `implements Observer` et la méthode `update` dans la définition de la classe `LView`).

Le lien entre objet observateur et objet observé se fait en "inscrivant" l'objet observateur dans la liste des observateurs de l'objet observé (appel de méthode `laby.addObserver(this)` dans le constructeur de `LView`). Quand l'objet observé `laby` signale un changement (appel `notifyObservers()`), le signal est transmis à tous les objets inscrits dans la liste des observateurs de `laby`, et donc en particulier à la vue, qui déclenche alors sa méthode `update()`. L'utilisation des classes et interfaces `Observer` et `Observable` de Java fait que nous n'avons presque rien à faire pour gérer ces mécanismes.

Le travail à effectuer est le suivant :

1. Faire que le héros ait accès à une méthode `void move(Direction dir)`, qui détermine la case où le héros doit se déplacer, qui retire le héros de sa case courante, et qui le place sur la nouvelle case. Vous devrez si ce n'est pas encore fait définir trois méthodes pour ces trois sous-tâches, dans la classe qui vous semblera adaptée. Dans un premier temps, disons que le héros peut traverser les murs comme les cases normales.
2. Dans le cas où le héros sort du labyrinthe, ce qui risque de lever une exception, faire que ladite exception soit rattrapée par le contrôleur, qui pourra alors afficher un message indiquant que le joueur a gagné.
3. Donner aux cases une méthode `public boolean isPassable()`, qui renvoie `true` si la case peut être traversée et `false` sinon. Les cases qui ne peuvent pas être traversées sont les murs et les cases occupées par des créatures. Les cases qui peuvent être traversées sont les cases normales et les cases de sortie (à condition qu'elles ne soient pas occupées par une autre créature).
4. Faire que votre méthode `move` du premier point lève une exception si vous essayez de déplacer le héros sur une case qui n'est pas traversable. Vous devrez définir vous-même cette exception. Faites que le contrôleur rattrape cette exception et affiche un message dans le terminal expliquant le problème (simplement avec `System.out.println`).
5. Faire que si le mouvement est impossible car la case cible est occupée par un monstre, alors une autre exception est levée, dont le rattrapage doit détruire le héros et afficher un message indiquant que le joueur a perdu.

N'oubliez pas de tester cette partie également après chaque point où cela est possible !

4 Animer les monstres

On veut que chaque déplacement du héros déclenche le déplacement de tous les monstres. Le déplacement de chaque monstre sera aléatoire, d'une case dans une des quatre directions ou en diagonale.

Le travail à effectuer est le suivant :

1. Ajouter au bon endroit une liste des monstres (de type `ArrayList<Monster>`), et faire que le constructeur de la classe `Monster` maintienne cette liste à jour.
2. Compléter l'énumération `enum Direction` pour y ajouter les directions nord-est, nord-ouest, sud-est, et sud-ouest. Modifier également dans cette classe la méthode `random()` pour qu'elle soit vraiment aléatoire.
3. Faire qu'une méthode `move` soit accessible aux monstres. Au besoin, réorganisez vos méthodes ou classes pour éviter les duplications de code entre les classes `Hero` et `Monster`.
4. Faire que le contrôleur fasse suivre chaque déplacement du héros par le déplacement de tous les monstres, et rattrape les exceptions qu'il y a à rattraper. N'oubliez pas que c'est au modèle d'effectuer ces déplacements, et qu'un signal doit indiquer à la vue de se mettre à jour. Si un monstre tente de se déplacer sur la case du héros, alors le joueur a perdu. En gérant bien les exceptions, vous pourrez aussi faire en sorte que si l'action du héros était impossible (mauvaise touche, ou direction bloquée par un mur), alors l'action n'est pas prise en compte et les monstres ne sont pas déplacés.

Pour tester cette partie, décommentez la méthode `putMonster` dans la classe `LModel`, et la troisième section de configuration dans la classe `Laby`.

5 Ajouter des portes

On veut que le labyrinthe contienne maintenant des cases spéciales qui seront des portes, qui peuvent être soit ouvertes soit fermées, et que le héros peut ouvrir et fermer. Une porte ouverte s'affichera en vert et sera une case traversable, alors qu'une porte fermée s'affichera en rouge et ne sera pas traversable.

Le travail à effectuer est le suivant :

1. Déterminer la stratégie de conception à utiliser pour gérer de bonne façon l'état ouvert ou fermé de vos portes.
2. Créer une classe `Door`, qui devra fournir deux méthodes de construction `openDoorFactory` et `closedDoorFactory` créant respectivement une porte ouverte et une porte fermée, et une méthode `changeState()` qui fait passer une porte de l'état ouvert à l'état fermé ou de l'état fermé à l'état ouvert.
3. Faire que les touches `y`, `n`, `j` et `g` déclenchent une tentative d'ouverture ou de fermeture de porte sur une case voisine du héros (respectivement au-dessus, en-dessous, à droite et à gauche). Comme aux parties précédentes, répartissez le travail de la bonne manière entre le modèle, la vue et le contrôleur.

Pour tester cette partie, décommentez les méthodes `putOpenDoor` et `putClosedDoor` dans la classe `LModel`, et la quatrième section de configuration dans la classe `Laby`.

6 Et ensuite

Vous pouvez ajouter ce que vous voulez à cette base ! D'ailleurs nous l'étendrons la semaine prochaine.