Rapport de projet d'informatique – POO
Réalisation d'un Snooker en java

INFO-H-301

DASNOY Sébastien
BALON-PERIN Alexandre
GEURY Thomas

Table des matières

1. Introduction

Ce projet consiste en la réalisation d'un snooker en langage de programmation java. Nous avons donc suivis les règles de l'orienté objet pour avoir un code le plus souple possible et qui puisse être facilement modifié pour des développements futurs.

Nous avons basé notre architecture sur le Pattern MVC (Modèle – Vue – Contrôleur) en divisant notre projet en différents packages. L'état du snooker changeant régulièrement, il faut faire communiquer ces packages entre eux pour qu'ils puissent se mettre à jour en échangeant des informations: c'est le rôle du Pattern Observer qui a été implémenté.

À l'intérieur de chaque package, nous avons également rigoureusement suivi les règles de l'orienté objet en utilisant les principes de l'héritage. En effet, lorsque plusieurs objets pouvaient être intuitivement généralisés en un objet plus général, nous en avait fait une super-classe, parfois abstraite. De plus, même lorsque l'on avait un seul objet qui, pour des développements futures, pouvait donner lieu à un autre objet avec des caractéristiques communes, nous avons fait appel au principe d'héritage.

Enfin, il était demandé d'avoir un code qui puisse facilement être modifié pour la réalisation d'un billard. Cela s'est fait très naturellement par respect des principes orientés objet.

2. Pattern MVC

La première étape de notre projet, après avoir réfléchi à toutes les considérations stratégiques présentées ci dessus, fut de composer l'architecture de notre code. Comme demandé, nous avons articulé nos classes autour du pattern MVC en les répartissant en trois packages: le modèle, la vue et le contrôleur. Chacun de ces packages a sa fonction propre:

- le modèle : il assure le bon déroulement du jeu et gère les modifications des paramètres en fonction des règles du snooker et d'un billard en général. . C'est donc la partie du programme qui se charge des calculs
- La vue: elle s'occupe uniquement de l'affichage en tenant l'utilisateur au courant de façon visuelle de l'état du modèle (et donc du jeu). Cela signifie que l'affichage sera modifié en fonction des nouvelles valeurs de paramètres des objets du modèle.
- Le contrôleur: il permet à l'utilisateur d'interagir avec le modèle. Cela signifie que les actions de l'utilisateur sont transmises au modèle par l'intermédiaire du contrôleur pour ensuite pouvoir être traitées. Dans le cas de ce snooker, l'utilisateur ne peut interagir que par l'intermédiaire de la souris pour bouger la queue ou la boule blanche.

Ensuite, après avoir implémenté ces différents packages, nous avons créé une classe *Principal.java* qui se charge d'organiser leur initialisation et de les relier les uns aux autres. Plus précisément, elle envoie le modèle au contrôleur pour que celui-ci, en récupérant les données transmises par l'utilisateur, puisse modifier les paramètres. Dans notre cas particulier du snooker, par exemple, le contrôleur détecte lorsque l'utilisateur recule la queue grâce à la souris et modifie son paramètre « position » en avertissant le modèle. Ensuite, le modèle doit communiquer cette modification à la vue pour que la queue recule à l'écran. Cela est réalisé grâce au pattern Observer, comme nous allons le voir au chapitre 3.

Dans le cadre de notre projet, le pattern MVC n'est pas réellement un avantage au niveau de l'efficacité puisqu'il ralentit le processus de déroulement du jeu en faisant circuler les données entre les différents packages (ce qui diminue la fluidité du mouvement des boules). En revanche, il apporte une réelle clarté dans la programmation et permet de facilement scinder l'implémentation du code. De plus, pour des développements futurs, il apporte une modularité au projet et est indispensable pour des projets plus conséquents.

3. Le contrôleur

Le package contrôleur sert de lien entre le modèle et l'utilisateur. C'est-à-dire que lorsque l'utilisateur bouge la souris, le contrôleur détecte le mouvement et l'interprète pour modifier le modèle. Les classes implémentées ici permettent donc de faire bouger la canne (et donc de démarrer un coup en déduisant du recul de la canne la vitesse à donner à la boule blanche) et de placer la boule blanche au début de la partie ou lorsqu'elle est tombée dans un trou. Ces deux classes utilisées (*BallMouseListener.java* et *QueueMouseListener.java*) implémentent les interfaces de gestion de la souris *MouseListener.java* et *MouseMotionListener.java.*

Pour améliorer la modularité de notre code, nous avons créé une sous-classe *BallMouseListenerSnooker.java* qui hérite de *BallMouseListener.java* pour y encapsuler les règles spécifiques au snooker et pouvoir éventuellement réaliser un jeu avec d'autres règles en minimisant les modifications à apporter. Cela était nécessaire car la boule blanche ne peut pas être positionnée au même endroit dans un billard ou un snooker par exemple. En revanche, les actions sur la queue donnent bien le même résultat qu'elles que soient les règles considérées, c'est pourquoi nous n'avons pas eu recours à l'héritage.

4. Pattern Observer

Comme précisé en introduction, nous avons utilisé le pattern Observer pour coupler les modifications des paramètres du modèle à la vue. Ainsi, dès qu'un paramètre du modèle (tel que la position d'une bille) est modifié par une action directe de l'utilisateur ou par le déroulement d'un tour, la vue en est averti et modifie l'affichage en conséquence.

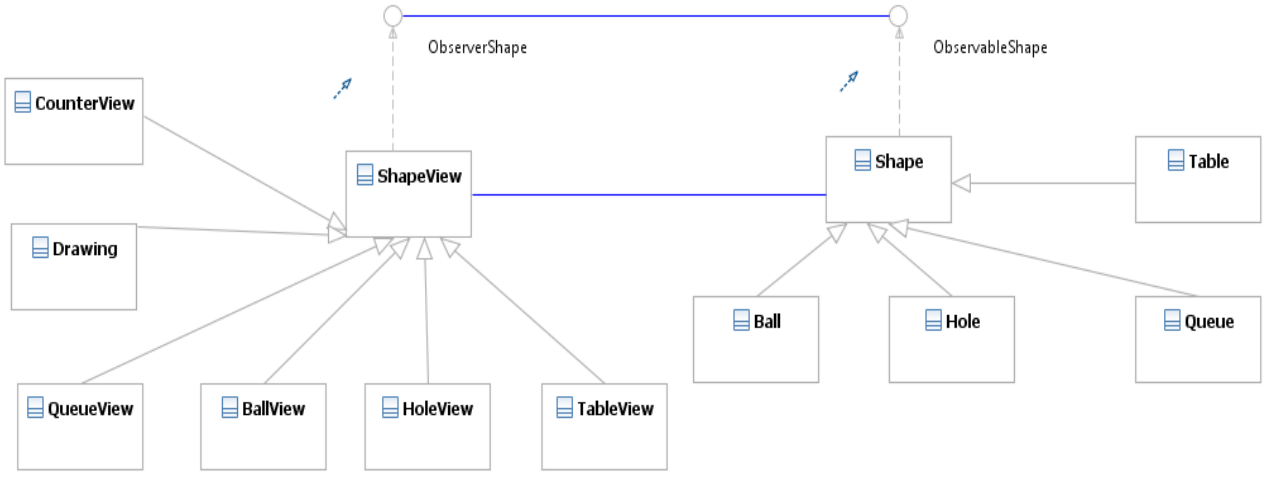
Ce pattern est donc utilisé pour tous les objets du package model à dessiner à l'écran, qui sont mis en relation avec leurs pendants du package view. Il fait donc le lien entre la classe *Shape.java* regroupant les objets à afficher et la classe *ShapeView.java* qui va mettre à jour l'affichage.

Pour ce faire, nous avons donc créé deux interfaces: *ObservableShape.java* et *ObserverShape.java* respectivement implémentées par *Shape.java* et *ShapeView.java*. Ainsi, lorsqu'un paramètre du modèle est modifié, on fait appel à la fonction notifyObserver implémentée par *Shape.java* qui prévient le ou les observer (ShapeView) qu'ils doivent se mettre à jour en appelant la fonction update que ShapeView implémente dans notre cas il y a un unique observer par objet. Ainsi, les objets affichables du package view (héritant de ShapeView, elle-même implémentant ObserverShape) sont informés des modifications des objets à afficher du package model. Lors de la création du lien entre chaque observer et observable une fonction nommée notifyObserverInit() est appelée et permet de donner au vue les valeurs des attributs tels que la taille des objets.

La finalité de cette communication détournée est de relier les deux packages par uniquement deux interfaces, ce qui a pour avantage de rendre le programme extrêmement modulable : tant que les informations communiquées le sont sous la même forme (mêmes types d'arguments), la mise à jour se fait, quelle que soit le fonctionnement du modèle et de la vue. En revanche, les liens nécessaires entre interface et classe ralentissent légèrement le fonctionnement du programme.

L'utilisation de ce Pattern dans ce programme n'est pas obligatoire, mais elle est fortement utile et permet de limiter le couplage. Le fait de séparer la vue du modèle permet aussi de modifier uniquement l'un ou l'autre package en gardant l'autre intacte ce qui peut s'avérer très pratique.

Une autre utilisation du pattern observer légèrement différente à lieu entre le compteur (Counter) du package model et le compteur (CounterView) du package view. Ainsi, lorsque le score d'un joueur est modifié, le compteur de la vue est averti et peut modifier l'affichage.



5. Le Modèle

Le package model contient toutes les classes de fond du programme. C'est grâce à ce package que les boules rebondissent entre elles, que les joueurs gagnent ou perdent des points lorsqu'ils empochent des boules... Il contient toutes les informations nécessaires aux calculs du programme et c'est donc lui qui se charge de modifier les paramètres.

Logiquement, ce package contient tous les objets physiquement contenus dans le snooker tels que la queue, les boules, la table, etc. Pour respecter les conventions de l'orienté objet, ces classes sont relativement générales et sont utilisées et modifiées dans d'autres classes qui gère le jeu. Comme précisé précédemment, pour faciliter le transfert de données avec la vue et étant donné que tous ces objets sont des formes ayant une certaine position dans la fenêtre, ils héritent tous de la classe *Shape.java.*

Parmi celles-ci, la classe abstraite ___*Game.*java gère le déroulement d'un tour de jeu indépendamment du jeu considéré. Elle est étendue par la classe __*Pool.java* qui est commune aux jeux de billard. Ainsi, cette dernière initialise les variables communes et gère un tour de jeu comme il pourrait l'être aussi bien dans un jeu de billard que de snooker. Enfin, en fonction du jeu considéré, il faut encore définir une sous-classe qui l'implémente en fonction des règles spécifiques. Dans notre cas, nous avons créé la classe *_Snooker.java* qui regroupe les caractéristiques de gestion du jeu spécifiques au Snooker.

La classe *RoundGestion.java* gère quand à elle le déplacement des billes durant un tour, leurs collisions entre-elles et avec la table. Elle hérite de la classe *Thread.java* et donc créé un thread ou processus qui permet de gérer le mouvement des boules.

Les autres classes notables sont les listes (PlayerList, BallsList, BallsListSnooker et HolesList). Toutes ces classes héritent de la classe liste préexistante de Java (ArrayList) à laquelle elles rajoutent des fonctions propres à l'utilisation des objets qu'elles contiennent. Par exemple, la liste des boules (BallsList) gère les rebonds entre les boules, et permet d'avoir un accès à la boule blanche. Sa classe fille spécifique au snooker (BallsListSnooker) a comme particularité supplémentaire de positionner les boules au début de la partie, et de replacer les boules de couleurs tombées dans les trous.

Enfin, remarquons que les interfaces observable/observer sont aussi dans le package model.

6. La vue

Le package view contient toutes les classes utiles à l'affichage de l'état du modèle. C'est grâce à ce package que l'utilisateur peut avoir connaissance des mouvements du jeu.

La classe principale du package view est Window. Elle contient la fenêtre dans laquelle la vue s'affichera. Cette fenêtre est organisée en deux parties distinctes : la table et tous les objets qui y sont affichés (Drawing), qui nécessite d'être rapidement actualisée, et le compteur (Counter), qui n'a besoin d'être actualisé qu'une fois les boules arrêtées. Cette séparation permet d'augmenter la fluidité du jeu.

A part les trois classes précédemment citées, toutes les classes implémentent la classe abstraite ShapeView, qui sert d'interface avec la classe Drawing pour les mise à jours de la table (et ShapeView implémente l'interface ObserverShape, qui permet aux objets à afficher d'être au fait de l'état du modèle).

7. Comment transformer le snooker en billard ?

Une des consignes était de limiter les changements à appliquer à ce snooker pour qu'il se transforme en billard. Cela traduit directement la modularité du code: si les concepts de l'orienté objet ont été rigoureusement respectés, la plupart des classes doivent rester indépendantes les unes des autres et donc il est possible « d'encapsuler » tout ce qui est spécifique à un snooker.

Dans notre code, seules trois sous-classes sont spécifiques au snooker: _S*nooker.java, BallsListSnooker.java* et *BallMouseListenerSnooker.java*. Elles étendent chacune une super-classe qui est commune au snooker et au billard. Cela signifie que pour réaliser un billard, il suffira de créer trois classes (*_Billard.java, BallsListBillard.java* et *BallMouseListenerBillard.java*) et de les implémenter en fonction des règles du billard. Les fonctions à implémenter sont clairement définies par les super-classes (*_Pool.java, BallsList.java et BallMouseListener*) et il ne faut donc plus réfléchir à l'architecture du code ni aller chercher dans tous les sens, il suffit de redéfinir ces fonctions en se concentrant sur les règles du billard.

Bien sûr, il faudra également modifier le lancement du jeu dans la classe *Principal.java*: il faut logiquement créer un billard à la place d'un snooker mais c'est la seule chose à changer.

N.B.: il aurait été possible de regrouper tout ce qui est spécifique au snooker dans une seule classe mais cela aurait nui à l'architecture orientée objet de notre programme et aurait surchargé cette unique classe du snooker (ex: toutes les fonctions de la classe *BallsListSnooker.java* auraient été implémentées dans la classe _S*nooker.java,* ce qui n'est pas très cohérent).

8. Conclusion

Ce projet nous a permis d'avoir un premier aperçu de la programmation orientée objet qui prévaut aujourd'hui dans le monde de l'informatique. En effet, nous avons essayé de nous en rapproché un maximum, notamment en développant notre code en parallèle avec notre diagramme UML pour assurer à notre projet une architecture cohérente. Bien sûr, nous aurions pu nous répartir le travail de manière plus systématique en développant chacun indépendamment une partie spécifique du projet pour ensuite pouvoir les assembler. Mais n'étant pas encore assez expérimentés dans le domaine, nous avons préféré travailler tous les trois en parallèle sur le code complet pour toucher à tout. Ce parallélisme fût grandement facilité par l'utilisation du plugin subclipse le subversion d'éclipse.

Au-delà de l'aspect programmation, nous avons appris à structurer un projet et le découper pertinemment pour plus de clarté et de modularité, ce qui nous sera certainement utile dans notre vie future d'ingénieurs car tout projet évolue et est sujet à des modifications et améliorations.

9. Remarques

1. Pour connaître le fonctionnement spécifique de chaque classe, se rapporter au code et aux commentaires de débuts de classes.
2. Un fichier « .jar » est également présent dans le projet pour permettre de lancer le programme en dehors d'Eclipse.
3. Des problèmes de chevauchement entre les billes persistent et sont sans doute dus à l'imprécision de la fonction « isOnShape ». Cependant, ils ne perturbent pas le déroulement du snooker dont les règles ont été rigoureusement respectées.
4. Les diagrammes UML du code sont aussi présent dans le projet éclipse.