ITI1521. Introduction à l'informatique II
Hiver 2016

Devoir 2

Échéance: 4 mars 2016, 23 h 59

[PDF]

Objectifs d'apprentissage

- Concevoir une application informatique avec le modèle de **programmation événementielle**
- Expliquez en vos propres mots le **patron de conception** modèle-vue-contrôleur
- Réaliser une application informatique en suivant le patron de conception **modèle-vue-contrôleur**

Information

Vous devez concevoir votre propre implémentation de l'application «circle the dot». Une version pour le système d'exploitation Android, de laquelle nous nous inspirons pour ce devoir, peut être téléchargée à partir de ce lien. Une version pour iOS peut être téléchargée en suivant ce lien.

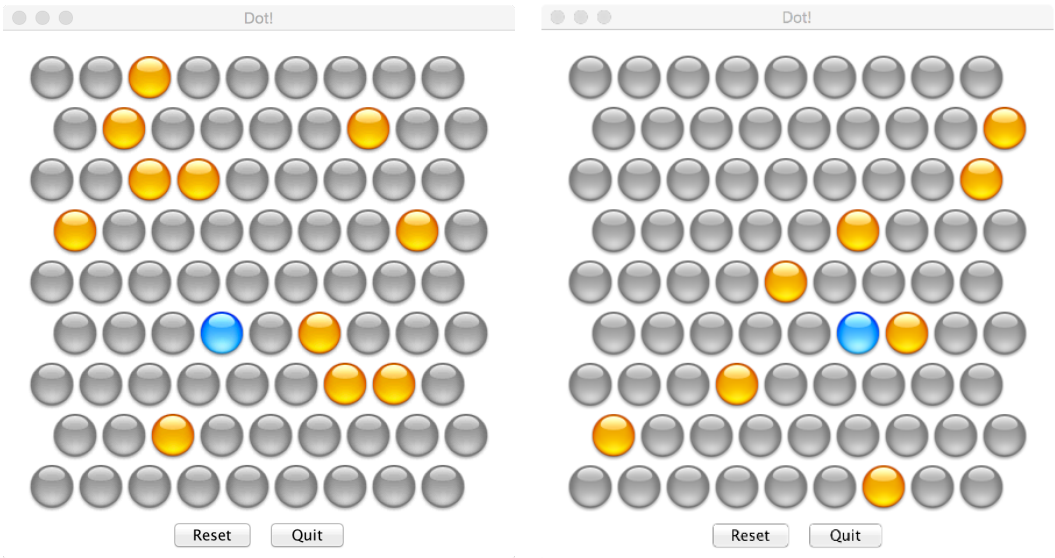

FIGURE 1 – Deux configurations initiales du jeu.

Le but du jeu est simple : une bille bleue cherche à s'évader de la grille de jeu, alors que le joueur lui cherche à l'empêcher en sélectionnant les billes grises, qui se transforment alors en billes oranges que la bille bleue ne peut franchir. La Figure 1 donne deux exemples de configurations initiales du jeu[1].

Le joueur gagne la partie si la bille bleue est encerclée de billes oranges (Figure 2). Comme vous pouvez le constater sur Figure 1, la configuration initiale comprend déjà des billes orange disposées aléatoirement sur la grille. La position initiale de

[1] L'interface est inspirée du jeu «Puzzler» d'Apple

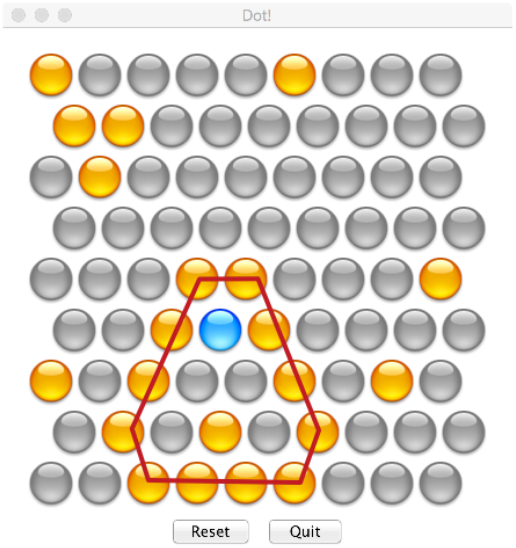

FIGURE 2 – Le joueur a gagné : la bille bleue est encerclée par les billes oranges.

la bille bleue est choisie aléatoirement parmi les positions centrales de la grille de jeu. Si la grille de jeu comprend un nombre pair de rangées et colonnes, alors la position de la bille bleue est choisie aléatoirement parmi les quatre positions centrales, alors que si le nombre de rangées et colonnes est impair, la position est choisie aléatoirement parmi les neuf positions centrales (Figure 3). À chaque étape du jeu, la bille bleue se déplace vers l'une des six positions voisines (Figure 4).

Modèle-vue-contrôleur

Le patron de conception («**design pattern**») **modèle-vue-contrôleur** (MVC — «Model-View-Controller») est utilisé fréquemment pour la conception d'interfaces utilisateur graphiques. Vous trouverez facilement des informations complémentaires sur Internet (notamment ici : Wikipedia, Apple, Microsoft, et Oracle, pour ne nommer que ceux-ci).

- Le **modèle** : ces objets représentent l'état de l'application.
- La **vue** (ou les vues[2]) : ces objets présentent le modèle à l'utilisateur (la portion visuelle de l'interface utilisateur graphique). Sa représentation reflète l'état courant du modèle. Il peut y avoir plusieurs vues simultanément. Cependant, pour ce devoir nous n'en aurons qu'une seule.
- Le **contrôleur** : ces objets implémentent la logique de l'application, comment l'état se transforme au cours de l'exécution en fonction des interactions externes (typiquement, les interactions avec l'utilisateur).

L'un grands avantages du patron de conception MVC est la séparation claire entre les différentes activités de l'application : le modèle ne représente que l'état courant de l'application, sans préoccupation pour son affiche ou encore les interactions avec l'utilisateur. La vue quant à elle n'est responsable que de la représentation (visuelle ou autre) de l'état du modèle. Notamment, la vue ne gère pas les interactions de l'usager. Finalement, le contrôleur est le «cerveau» de l'application. Il ne se préoccupe pas de représenter l'état de l'application ou son interface utilisateur.

En plus de cette séparation claire, MVC fournit un schéma de collaboration logique entre les composantes de l'application (Figure 5). Voici le schéma de collaboration pour ce devoir :

1. Lors d'une interaction avec la vue (pour ce devoir, lorsque l'utilisateur sélectionne une bille grill), le contrôleur en est informé (message 1 de la Figure 5).
2. Le contrôleur traite l'information et met à jour le modèle de façon appropriée (message 2 de la Figure 5).

[2]C'est l'une des grandes forces de ce patron de conception, on peut facilement associer plusieurs vues à l'application, par exemple, on peut associer une vue pour les usagers ayant un handicap visuel.

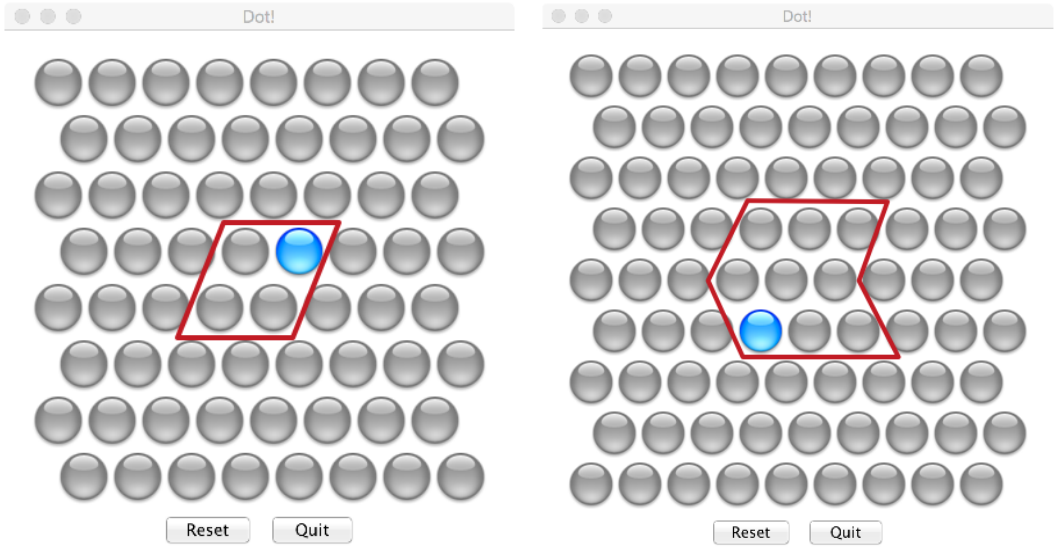

FIGURE 3 – Position initiale de la bille bleue dans le cas où le nombre de rangées et colonnes est pair (à gauche). Dans ce cas, la position de la bille est sélectionnée aléatoirement parmi les quatre positions centrales. Dans le cas où le nombre de rangées et colonnes est impair (à droite), la position initiale de la bille bleue est choisie parmi les neuf positions centrales.

3. Une fois l'information traitée et le modèle mis à jour, le contrôleur informe la vue (ou les vues) qu'elle doit être rafraîchie (message 3 de la Figure 5).
4. Finalement, chaque vue doit consulter le modèle afin de refléter fidèlement l'état courant (message 4 de la Figure 5).

1 Modèle [20 points]

La première étape sera de construire le modèle du jeu. Pour ce faire, nous utilisons deux classes, **GameModel** et son compagnon **Point**. Au cours de l'exécution du jeu, il n'y aura qu'une seule instance de la classe **GameModel**. Cet objet mémorise l'état courant du jeu :

- Définition de l'état d'une bille. Chaque bille a trois états possibles :
  - **AVAILABLE** : la bille n'est pas bleue et n'a pas été sélectionnée.
  - **SELECTED** : la bille n'est pas bleue et a été sélectionnée.
  - **DOT** : c'est la bille bleue.
- La position actuelle de la bille bleue.
- L'état de chaque bille sur la grille du jeu.
- Le nombre d'étapes depuis de début du jeu.
- La taille de la grille du jeu.

Le modèle fournit les méthodes d'accès nécessaires («setters» et «getters») pour que le contrôleur et la vue aient facilement accès à l'état de chaque bille, mais aussi pour que le contrôleur puisse changer l'état d'une bille et déplacer la bille bleue. Finalement, le modèle a une méthode pour initialiser le jeu : placer aléatoirement la bille sur la grille selon les explications ci-dessus, et présélectionner certaines billes selon une probabilité préétablie. Dans notre cas, chaque bille a 10% de chance d'être présélectionnée.

Une description détaillée des classes se trouve ici :

- JavaDoc pour GameModel
- GameModel.java

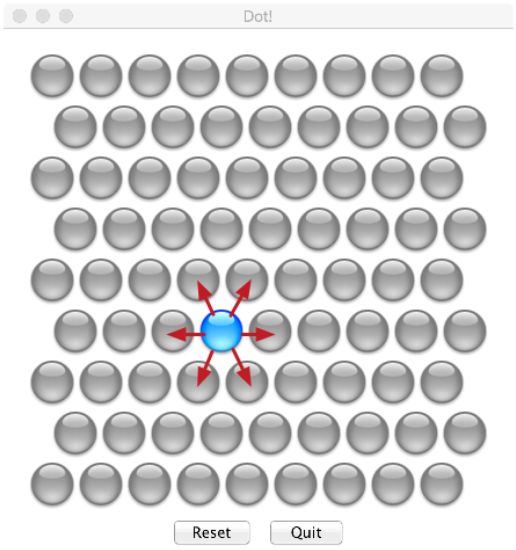

FIGURE 4 – La bille bleue se déplace d'une position à la fois, sur l'une des six positions adjacentes.

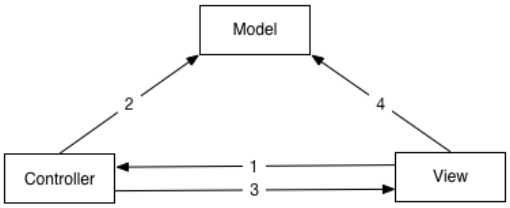

FIGURE 5 – La collaboration entre le **modèle**, la **vue**, et le **contrôleur**

- JavaDoc pour Point
- Point.java

2 Vue [30 points]

Vous devez construire l'interface utilisateur graphique du jeu. Pour ce faire, vous utiliserez deux classes. La classe **GameView** doit être une sous-classe de la classe JFrame. C'est la fenêtre principale de l'application. Cette classe affiche deux boutons au bas de la fenêtre, remise à zéro et un bouton pour quitter le jeu. Elle inclut aussi une instance de la seconde classe, **BoardView**.

La classe **BoardView**, une sous-classe de JPanel, représente la grille du jeu. La grille est constituée de n rangées et n colonnes, n représente ici la taille du jeu. Chaque cellule renferme une bille. Pour implémenter les billes, nous utilisons la classe **DotButton**, une sous-classe de JButton. **DotButton**. **DotButton** est fortement inspirée de l'application **Puzzler** présentée au laboratoire 4. La version **Java** de l'application **Puzzler** est une réécriture de l'application Puzzler développée par Apple.

La difficulté principale de l'interface graphique utilisateur est la représentation de la grille. Si la grille était un carré régulier, tel que représenté par la Figure 6, il serait facile de déposer les objets **DotButton** sur la grille. Une simple utilisation du gestionnaire GridLayout suffisait. Cependant, la grille de notre application n'est pas régulière (Figure 7). Les lignes sont décalées vers la gauche ou vers la droite. Il n'y a donc pas de solution immédiate.

Nous vous proposons une approche en deux étapes : chaque rangée (d'objets **DotButton**) aura son propre objet **JPanel** et utilisera le gestionnaire FlowLayout. Si vous consultez la documentation de la classe JPanel, vous verrez qu'on peut ajouter une bordure au panneau. Puisque les objets **DotButton** ont une dimension de 40 pixels, l'idée consiste à ajouter une bordure de 20 pixels au bon endroit afin d'obtenir le rendu visuel désiré. Les rangées (objets de la classe **JPanel**) sont ajoutées à un objet **JPanel** englobant dont le gestionnaire sera de type **GridLayout**.

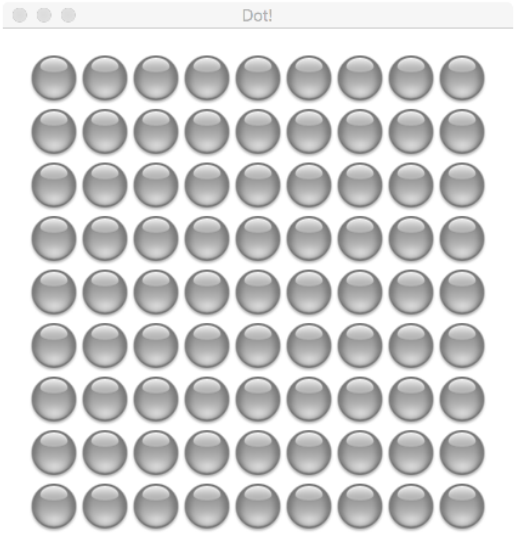

FIGURE 6 – Une grille parfaitement carrée.

Bien que **GameView** et **BoardView** comprennent des boutons, ces deux classes ne sont pas les gestionnaires («*listeners*») des événements générés par ces boutons. Ce sera la tâche du contrôleur (voir Question 3).

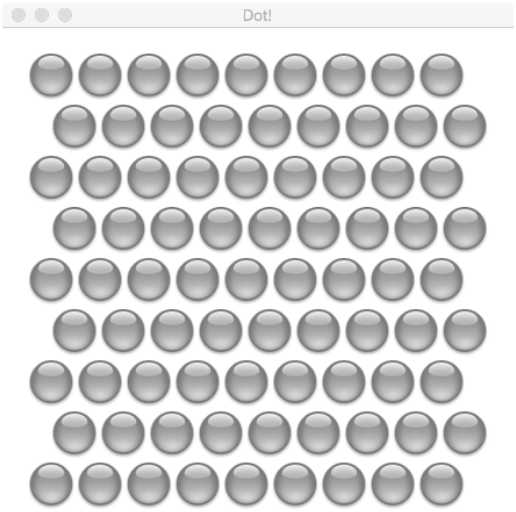

FIGURE 7 – La représentation réelle de la grille.

Une description détaillée des classes se trouve ici :

- JavaDoc pour GameView
- GameView.java
- JavaDoc pour BoardView
- BoardView.java
- JavaDoc pour DotButton

- DotButton.java
- Un fichier zip contenant tous les icônes. Tirée de l'application Puzzler d'Apple.

3 Contrôleur [40 points]

Finalement, vous devez implémenter le contrôleur. C'est la classe **GameController**. Le constructeur de la classe reçoit en paramètre la taille de la grille de jeu. Une instance du modèle et une instance de la vue sont créées par ce constructeur.

L'instance de la classe **GameController** gère les événements résultant des interactions avec l'utilisateur. Cette classe doit donc implémenter les méthodes nécessaires, et la logique du jeu. En cours de jeu, lorsque l'usager sélectionne une nouvelle bille, le contrôleur doit décider la prochaine action à prendre. Il y a trois cas possibles : le joueur a gagné, le joueur à perdu, ou sinon la bille bleue doit se déplacer vers une position adjacente afin d'éviter d'être encerclée.

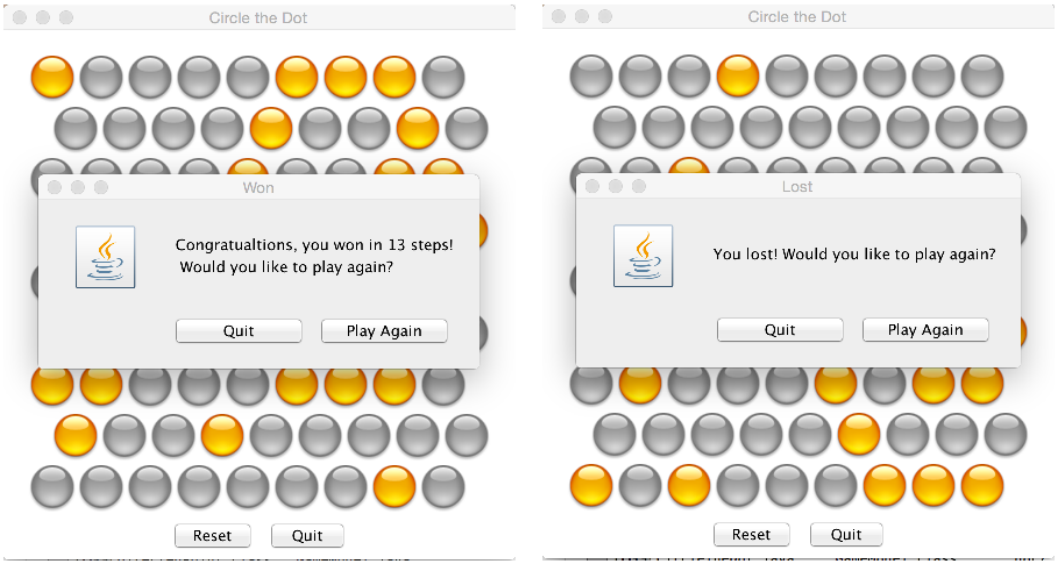

FIGURE 8 – Gagné et perdre un jeu.

Dans les deux premiers cas, le contrôleur doit informer l'utilisateur du résultat. Dans le cas d'une victoire, il doit aussi afficher le nombre d'étapes menant à la victoire (Figure 8). Pour y arriver, le contrôleur utilisera la méthode de classe **showOptionDialog** de la classe JOptionPane.

Si le jeu se poursuit, vous devez implémenter une solution efficace pour la sélection du prochain déplacement de la bille bleue. Nous vous proposons deux stratégies :

- La bille bleue se déplace aléatoirement vers une position voisine non sélectionnée. Cette solution est facile à implémenter et vous permettra de compléter votre application rapidement. Vous n'obtiendrez que des points partiels, si c'est votre seule et unique solution finale.
- La bille se déplace dans la direction du plus court chemin permettant à la bille bleue de s'échapper de la grille du jeu.

Fouille en largeur

Pour trouver le court chemin, nous utiliserons une implémentation de l'algorithme de fouille en largeur («breadth-first-search»). Cet algorithme sera utilisé afin de trouver le plus court chemin entre la position courante de la bille bleue et le rebord de la grille de jeu. Comme nous le verrons bientôt en classe, cet algorithme est assuré de toujours trouver le plus court chemin. Nous verrons aussi que l'utilisation d'une file («queue») est l'une des façons les plus simples d'implémenter l'algorithme de fouille en largeur.

Pour implémenter notre file, nous utiliserons une instance de la classe LinkedList. Afin d'ajouter un objet à la file, nous utiliserons sa méthode **addLast**, et pour retirer un élément, nous utiliserons sa méthode **removeFirst**.

Voici une description à haut niveau de la fouille en largeur implémentée à l'aide d'une file. La fouille débute à la position départ («start») et cherche le chemin le plus court vers l'une des positions de la liste de cibles («targets»). L'algorithme a aussi à sa disposition une liste de positions proscrites, ce sont des positions que la bille bleue ne peut occuper («blocked»). Initialement, la liste «blocked» ne contient que les positions des billes sélectionnées. Durant la recherche, la liste «blocked» contiendra aussi les cellules visitées par le chemin (afin d'éviter de tourner en rond). Voici le pseudo-code de l'algorithme. Deux positions sont voisines si l'on peut passer d'une position à l'autre en seul déplacement.

```
Breadth-First-Search(start, targets, blocked)

 create an empty queue.

 add the path "{start}" to the rear of the queue.

 While the queue is not empty Do
   Remove the path q from the front of the queue. Let c be the last
         position of that path
   For all positions p neighboring c
     If p is not in blocked Then
       If p is in targets then
         return the path q + {p}
       Else
         add the path q + {p} to the end of the queue.
         add p into blocked
       End If
     End If
   End For
 End While

 // The queue is empty and we have not returned a path. The targets
 // are not reachable from position start.
 return FAIL
```

CircleTheDot

Une instance de la classe **GameController** est créée par la méthode principale de la classe **CircleTheDot**. Un paramètre d'exécution est passé à la méthode principale (**main**) afin de spécifier la taille de la grille (taille minimum de 4). Si la valeur du paramètre n'est pas valide, la valeur par défaut, 9, est utilisée.

Une description détaillée des classes se trouve ici :

- JavaDoc pour GameController
- GameController.java
- JavaDoc pour CircleTheDot
- CircleTheDot.java

Bonus [10 points]

Le pseudo-code de l'algorithme de fouille en largeur est déterministe. Un joueur astucieux pourrait anticiper le prochain déplacement de la bille bleue. Modifiez l'algorithme afin que le prochain déplacement soit choisi parmi tous les plus courts chemins possible (il s'agit bien du plus court chemin, mais il est choisi aléatoirement parmi tous les plusieurs chemin, s'il y en a plus d'un).

Respect des règles et consignes [5 points]

Veuillez suivre les consignes que vous trouverez sur la page des consignes aux devoirs. Tous les devoirs doivent être soumis à l'aide de `https://uottawa.blackboard.com`. Vous devez préférablement faire le travail en équipe de deux, mais vous pouvez aussi faire le travail seul. Vous devez utiliser les gabarits qui vous sont fournis ci-dessous.

Fichiers

Vous devez soumettre un fichier zip contenant les fichiers suivants :

- Un fichier texte nommé **README.txt**, contenant le nom des coéquipiers, leurs numéros d'étudiants, la section du cours, ainsi qu'une courte description du devoir (une ou deux lignes).
- Le code source de **toutes** les classes.
- Un répertoire **data** avec toutes les images.
- Un répertoire JavaDoc
- StudentInfo.java, avec vos informations dûment complétées, ainsi qu'un appel à la méthode **display** à partir de votre méthode principale.

Comme toujours, nous devons être en mesure de compiler et exécuter votre application après avoir extrait les fichiers de l'archive zip.

Modifié le 10 février 2016