27 février 2024

PROGRAMMATION ORIENTÉE OBJET

PROJET 2

PROGRAMMATION CONCURRENTIELLE ET ÉVÈNEMENTIELLE DANS LE PATRON DE CONCEPTION MVC D'UN JEU DE PUISSANCE 4 DOTÉ D'UNE IA MIN-MAX

Table des matières

Présentation du projet

Ce projet est le développement d'un programme permettant aux utilisateurs de jouer des parties de puissance 4 entre eux, contre une intelligence artificielle (*IA*) de type *minmax*, ou encore d'observer une partie disputer entre deux telles *IA*.

Comme les lacunes soulevées dans le premier projet de *POO* évoquaient principalement les bonnes pratiques, je me suis concentré sur cet aspect lors de la réalisation de ce deuxième projet.

Pour ce faire, je me suis soucié de concevoir une architecture logicielle compréhensible, flexibles et maintenable, en tâchant de respecter les principes de l'acronyme S.O.L.I.D., en suivant le patron de conception modèle-vue-contrôleur (*MVC*) et en commentant le code avec la plus grande précision.

L'étude du patron de conception *MVC* m'a naturellement conduit à apprendre les rudiments de l'interface graphique, de la programmation concurrentielle et de la programmation évènementielle.

Organisation et fonctionnement

Généralités

Conformément au patron de conception *MVC*, le projet se compose d'une classe *Main* et de trois répertoires comportant les paquetages *model* pour le modèle, *view* pour la vue et *controller* pour le contrôleur.

Plus précisément, le répertoire *model* contient toutes les classes, interfaces et énumérations représentant les données relatives aux parties de *puissance 4*, le répertoire *view* contient toutes celles traitant de l'interface graphique, et le répertoire *controller* se compose de l'unique classe `Controller` chargée d'orchestrer le déroulement du programme en gérant les interactions entre le modèle et la vue.

La définition de la classe `Main`, unique classe appartenant au répertoire du projet, est l'occasion de comprendre la notion de programmation concurrentielle. En effet, le processus engendré par le programme est en fait composé de plusieurs sous-processus, appelés aussi files d'exécution (ou *thread*), évoluant en concurrence dans le sens ou leurs requêtes auprès des processeurs se font indépendamment, et donc de façon asynchrone. Cette classe crée un *thread* (file d'exécution) qui implémente l'interface `Runnable` dont l'unique méthode `run()` contient le code à exécuter par ce dernier. Il crée une instance de classe `Model`, une autre de classe `View` et une dernière de classe `Controller`, centralisant respectivement toutes les ressources de chacun des paquetages *model*, *view* et *controller*. Il fixe pour finir les règles régissant les interactions entre les trois entités de ce patron de conception en appelant la méthode `observedBy(View)` de l'instance de `Model` avec pour paramètre l'instance de `View` puis en appelant la méthode `listenedBy(Controller)` de l'instance de `View` avec pour paramètre l'instance de `Controller`. Ces deux derniers appels rendent certains objets (dont les créations ont été induites par les constructeurs `Model()`, `View(Model)` et `Controller(Model,View)`) réactifs aux actions des utilisateurs via la souris et aux changements de valeurs de certains attributs. Chacune de ces actions et chacun de ces changements de valeur est en fait associé à une entité du point de vue du système d'exploitation appelée évènement. Comme les instructions traduisant la réactivité susmentionnée sont exécutées dans un autre *thread* appelé le « *Event Dispatch Thread* » (*EDT*), on parle de programmation concurrentielle.

La figure suivante montre qu'il y a deux types de relation entre les trois entités du patron de conception *MVC*, la connaissance et l'observation (ou l'écoute) :

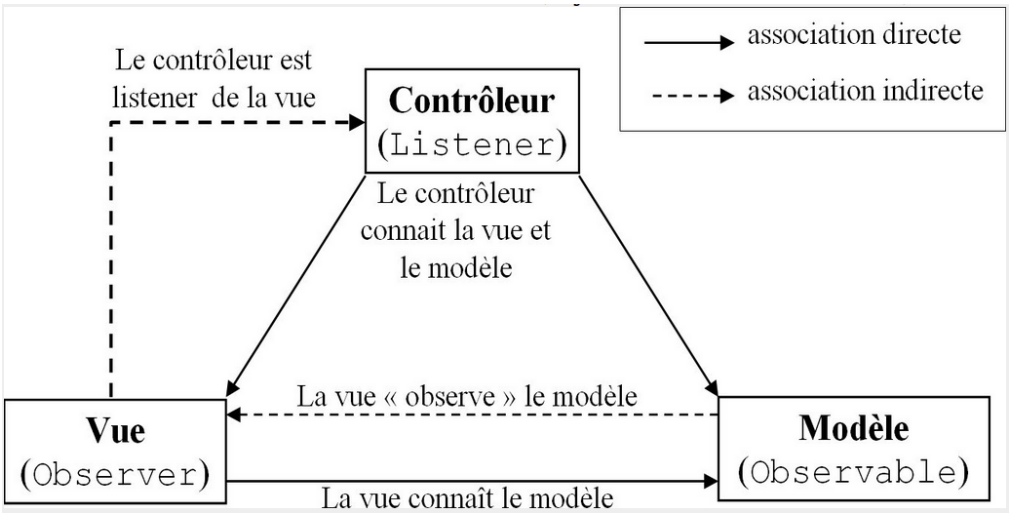

FIGURE 1 – schéma de l'architecture MVC extrait de la section « Une conception sur le modèle MVC (Model-View-Controller) » du cours de *Java* d'Irène Charon sur perso.telecom-paristech.fr

Les relations de connaissance sont implémentées en rendant les instances de `View` et de `Model` attributs de l'instance de `Controller`, en rendant l'instance de `Model` attribut de `View`, et en ne faisant pointer aucun des attributs de l'instance de `Model` sur aucune donnée relative aux deux autres entités. En d'autres termes, ces déclarations et non déclarations de

pointeur font que le contrôleur connaît la vue et le modèle, la vue connaît le modèle, et le modèle ne connaît aucune des deux autres entités.

Mais ces relations de connaissance sont insuffisantes car elles ne permettent pas de déclencher l'exécution les blocs d'instructions qui suivent immédiatement chacun des différents moments clefs du processus, c'est-à-dire les blocs d'instructions correspondant à chacun des évènements. En effet, ces derniers ne sont pas exécutés par le *thread* créé par la classe `Main` et la machine virtuelle *Java*. Ils sont exécutées par le *thread EDT* en réponse aux évènements évoqués précédemment. C'est pourquoi on parle de programmation évènementielle.

Les relations d'observation s'implémentent alors en déclenchant des évènements et en réagissant à des évènements selon le tableau ci-dessous :

cas de déclenchement d'évènement	méthode appelée en réponse à l'évènement
sélection de l'item « nouvelle partie » dans le menu « nouveau »	`actionPerformed` de l'instance de `Controller` qui appelle en particulier la méthode `reset` de `Model` pour réinitialiser le modèle
sélection de l'item « jouer entre humains à la prochaine partie » dans le menu « paramètres »	`itemStateChanged` d'une instance anonyme de `ItemListener` créée par le constructeur de l'instance de `View` pour désactiver le groupe de boutons relatif au choix de l'utilisateur de commencer ou non la prochaine partie contre l'*IA*
sélection de l'item « jouer contre l'IA à la prochaine partie » dans le menu « paramètres »	`itemStateChanged` d'une instance anonyme de `ItemListener` (la même que celle mentionnée ci-dessus) pour activer le groupe de boutons relatif au choix de l'utilisateur de commencer ou non la prochaine partie contre l'*IA*
clic de souris d'un utilisateur sur le panneau représentant le dessin de la grille de jeu	`mouseClicked` de l'instance de `Controller` pour mettre à jour l'objet de classe `EvolutiveGridParser` représentant la grille de jeu et les données déduites correspondant à un coup joué de sa part dans la colonne survolée par la souris sur le panneau représentant la grille de jeu
modification de l'attribut `cellState` d'un objet de classe `Cell` représentant la couleur d'une cellule de la grille de jeu	`propertyChange` de l'instance de `View` pour déclencher l'animation de la chute du jeton correspondant à l'instance de `Cell`
appel à la méthode `stop` de l'attribut `fallingTokenTimer` de l'objet représentant le dessin de la grille de jeu marquant la fin de l'animation représentant la chute d'un jeton	`propertyChange` de l'instance de `Controller` pour mettre à jour les éventuelles instances de `AIPlayer` représentant les joueur *IA* et faire jouer le prochain joueur si c'est une *IA* ou, dans le cas contraire, permettre à un utilisateur de jouer le prochain coup via un clic de souris en rendant réactif le panneau de l'interface graphique représentant le dessin de la grille de jeu
appel à la méthode `reset` correspondant la réinitialisation de l'instance de `Model` (qui est appelée en conséquence indirecte à la sélection de l'item « nouvelle partie » mentionnée dans la première ligne de ce tableau)	`propertyChange` de l'instance de `View` pour réinitialiser l'attribut représentant le dessin du jeu sur l'interface graphique, ce qui traduit le commencement d'une nouvelle partie

Modèle

Les attributs de la classe `Model` représentent les deux couleurs de jeton, les deux joueurs, un objet de classe `EvolutiveGridParser` représentant la grille de jeu et ses données déduites constantes et évolutives, et un objet de classe `PropertyChangeSupport` chargé de déclencher un évènement pour que l'instance de `View` réinitialise l'attribut représentant le dessin du jeu sur l'interface graphique, ce qui traduit le commencement d'une nouvelle partie (voir la dernière ligne du tableau précédent).

Les instances des classes `HumanPlayer` et `AIPlayer` héritant de la classe abstraite `AbstractPlayer` représentent respectivement les joueurs utilisateurs et les joueurs *IA*.

L'attribut de classe `EvolutiveGridParser` de l'instance de `Model` hérite de la classe `ConstantGridParser` regroupant les données déduites constantes de la grille de jeu, qui hérite à son tour de la classe `Grid` représentant simplement la grille de jeu.

La classe `Grid` est en fait une sous-classe de la classe générique `HashSet<Cell>`. Les objets de classe `Cell` modélisant les cellules de la grille de jeu, chacune des instances de classe `Grid` représente donc l'ensemble des cellules d'une grille de jeu.

Les attributs de la classe `Cell` représentent les coordonnées des cellules sur la grille de jeu et leur couleur. C'est le rôle de la classe `Coordinates` et de l'énumération `CellState` que de créer de telles attributs.

Il est à noter dès à présent qu'une partie, lorsqu'elle implique au moins un joueur *IA*, ne fait pas intervenir qu'une seule instance de `EvolutiveGridParser`. En effet, un objet de classe `AIPlayer` représentant un joueur *IA* comporte un attribut représentant un arbre de morceaux de parties possibles depuis la position courante lui permettant d'y appliquer l'algorithme du *min-max*. Chacune des intersections de branche est alors associée à un objet de classe `EvolutiveGridParser` pour représenter et évaluer une position potentielle. Pour optimiser le mémoire virtuelle qui est alors vouée à accuser une progression géométrique de combinaison inhérente à la structure d'arbre, ces copies de classe `EvolutiveGridParser` ne sont pas intégrales. Elles sont le résultat d'un clonage partiel ne recopiant pas les instances de `Coordinates` représentant les coordonnées des cellules sur la grille de jeu. C'est la classe statique `CloneWithSameCoordinatesFactory` qui est chargée de ce clonage.

Les attributs de la classe `AIPlayer` représentent l'infini (via le nombre 10^{20}) pour pondérer fortement les positions gagnantes lors de l'évaluation, leur arbre des morceaux de parties possibles depuis la position courante, et la profondeur de ce dernier. Cette classe contient deux méthodes dites « de jardinage » dans les commentaires du code dont l'une est chargée d'élaguer l'arbre lors de la mise à jour correspondant à la prise de connaissance du dernier coup joué et l'autre est chargée de faire pousser l'arbre d'une génération de branche afin qu'il retrouve sa profondeur initiale. Les méthodes chargées de l'évaluation valorisent le nombre de groupes de quatre cellules alignées (classe `FourCellsInARow`) constituées de cellules non encore jouées ou de la couleur jouée par le joueur *IA*, le nombre de leurs cellules colorées (de la couleur du joueur *IA*), et la moyenne des altitudes dans la grille de jeu de leurs couleurs non encore jouées.

La classe `IATree` instancie les arbres de morceaux de parties possibles. Leurs attributs représentent le parseur évolutif de grille représentant l'état de la grille correspondant à la racine, la couleur du prochain coup pour cette dernière position, la profondeur de l'arbre, un booléen indiquant si l'arbre est une feuille, c'est-à-dire s'il n'est constitué que de sa racine (position nulle, gagnante ou située à la limite définie par la profondeur attribut des instances de la classe `AIPlayer`), et un dictionnaire qui, à une instance de l'énumération `ColumnNumber` représentant un numéro de colonne, associe une autre instance de `AITree` représentant la première génération de branches et conférant à cette dernière classe une structure d'arbre.

Vue

La vue, l'instance de classe `View`, écoute le modèle, l'instance de `Model`, dans le sens où elle implémente la méthode `propertyChange` de l'interface `PropertyChangeListener` qui s'exécute lors du changement de la couleur d'une cellule de la grille de jeu ou lors de la réinitialisation de la grille de jeu. C'est ainsi que la vue reflète fidèlement l'état du modèle à l'utilisateur à chaque instant via une interface graphique. Elle est aussi chargée de capter les actions des utilisateurs afin que le contrôleur, écouteur de la vue, en soit informé via cette même stratégie évènementielle.

L'instance de classe `View` hérite en fait de la classe `JFrame` pour correspondre à une fenêtre graphique visible par les utilisateurs. Outre l'instance de `Model` qu'elle doit connaître pour la refléter, elle dispose d'attributs établissant une barre de menu via laquelle l'utilisateur peut régler les paramètres déterminant la nature des joueurs de la prochaine partie et lequel va la commencer, lancer une partie disputée par deux joueurs *IA*, ou commencer une nouvelle partie pour affronter un autre utilisateur ou un joueur *IA*. Un autre attribut de classe `JLabel` offre une zone de texte dans la partie supérieure de la fenêtre dans laquelle s'affiche en particulier les notifications en provenance du contrôleur (c'est le seul cas de transmission directe du contrôleur vers le modèle).

Un dernier attribut de l'instance de `View` de classe `Drawing` héritant de la classe `JPanel` est le plus important car il constitue un dessin de la grille de jeu interactif dans la partie inférieure de la fenêtre. C'est sa méthode `paintComponent` qui est chargée de dessiner ce dessin. Elle appelle d'abord la méthode du même nom de la classe `JComponent` dont hérite la classe `JPanel`, transtype son paramètre de la classe `Graphics` à la classe `Graphics2D` afin de récupérer les méthodes traitant des implémentations de l'interface `Shape` de la bibliothèque *AWT* qui permettent ensuite de dessiner l'attribut `paintableGrid` de classe `PaintableGrid` représentant la grille de jeu sans les jetons puis, à tour de rôle, via une boucle `for`, les objets implémentant l'interface `Drawable` que contient l'attribut `setOfDrawableNotPaintableGrid` de classe `HashSet<Drawable>` de la classe `Drawing`.

Les objets à dessiner respectent une hiérarchie des interfaces. Ils héritent tous de l'interface `Drawable` constituée de la seule méthode `draw` et qui s'étend en deux sous-interfaces `Paintable` et `Colorable` implémentées par les objets qui se dessinent en remplissant une instance implémentant `Shape` respectivement avec une image et une couleur.

Les deux attributs de classes `PaintableGrid` et `PaintablePossibleNextToken` de l'instance `Drawing` et qui implémentent l'interface `Paintable` représentent respectivement la grille de jeu sans les jetons et l'éventuel prochain jeton qui serait joué dans la colonne de la grille de jeu survolée par la souris. Les images associées sont contenues dans les fichiers *game.png* et *question_mark.png*. Avant le remplissage de la forme associée à `PaintableGrid`, l'image est redimentionnée via la méthode `resize` de la classe statique `ImageManager`. La forme de la grille, i.e. l'attribut de l'instance de `PaintableGrid` de classe `Area` qui implémente l'interface `Shape`, est aussi trouée par des disques (de classe `Ellipse2D.Double`) avant le remplissage via sa méthode `subtract` afin de ne pas cacher le dessin des jetons. La seule classe fille de l'interface `Colorable` est la classe `ColorableToken` et instancie les objets qui représentent les jetons réellement joués. Ces derniers sont donc simplement colorés et non peints avec une image.

Tous ces éléments graphiques s'organisent également en une hiérarchie des classes suivant leur forme. Ils héritent tous de la classe abstraite `DrawingElement`. La seule classe concrète qui en hérite directement est la classe `PaintableGrid`. Les autres classes représentant les éléments graphiques héritent des sous-classes `DrawingDisk` et `DrawingRectangle` de `DrawingElement`. Les classes filles de `DrawingDisk` sont `ColorableToken` et `PaintablePossibleNextToken`. La classe `InvisibleColumnArea` est l'unique classe fille de la classe `DrawingRectangle` et instancie des objets chargés de simplifier la localisation de la souris

pour connaître sur quelle colonne de la grille de jeu l'utilisateur cherche à jouer. Ils correspondent effectivement à chacune des zones recouvrant une colonne de la grille de jeu sur le dessin généré par l'instance de `Drawing`.

L'attribut `fallingTokenTimer` de classe `ListenableTimer` de l'instance de `Drawing` permet les animations représentant la chute des jetons dans la grille de jeu. En effet, la classe `ListenableTimer` hérite de la classe `Timer` dont les méthodes `start` et `stop` activent et désactivent respectivement la création régulière d'évènements de classe `ActionEvent` qui provoquent chacun l'exécution de la méthode `actionPerformed` d'un écouteur de classe `ActionListener`. Dans le cas présent, l'écouteur est l'instance de `Drawing` et les instructions de sa méthodes `actionPerformed` décalent verticalement le dernier jeton joué représenté par l'attribut `fallingToken` de l'instance de `Drawing`. L'extension de la classe `Timer` à la classe `ListenableTimer` est l'ajout d'un attribut de classe `PropertyChangeSupport` qui déclenche un évènement de classe `PropertyChangeEvent` lors de l'appel à la méthode `stop` ré-implémentée de la sorte pour que le contrôleur, en écoutant en particulier les évènements de cette classe, soit informé de la fin de l'animation représentant la chute d'un jeton afin qu'il puisse reprendre la main en exécutant sa méthode `propertyChange`.

Contrôleur

Le contrôleur, l'instance de classe `Controller`, écoute la vue dans le sens où il implémente les méthodes des interfaces `MouseListener`, `ActionListener` et `PropertyChangeListener` qui sont exécutées lorsque des évènements correspondant aux actions de l'utilisateur par le biais de l'interface graphique ou correspondant au changement de la valeur de certains attributs de la vue.

Ces méthodes modifient le modèle, l'instance de `Model`, et notifie la vue, l'instance de `View`.

Problèmes rencontrés

L'apprentissage du patron de conception *MVC* a demandé beaucoup de temps, notamment en ce qui concerne les relations « connaître » et « observer » entre les différentes entités. De plus, la littérature présente plusieurs type de *MVC* selon les interactions entres le modèle, la vue et le contrôleur. Il a donc fallu réfléchir pour en choisir un qui s'adapte convenablement au problème.

Les interfaces `Observer` et `Observable` dont le rôle est d'implémenter les relations d'observation susmentionnées sont dépréciées car elles ne permettent pas de transmettre d'informations précises concernant ce qu'elles signalent. Un travail de recherche conséquent a alors été nécessaire pour penser à remplacer cette stratégie par l'interface `PropertyChangeListener` et la classe `PropertyChangeSupport`. Il a donc fallu aussi étudier la norme *bean*.

Ce travail a également exigé l'apprentissage des paquetages et d'étudier plusieurs cours et tutoriels sur les bibliothèques graphiques *AWT* et *SWING* afin d'élaborer une interface graphique.

De nombreuses heures ont alors dues être consacrées à l'apprentissage des rudiments de la programmation évènementielle et de la programmation concurrentielle indispensables à la compréhension du dispositif de l'interface graphique.

Pour respecter les bonnes pratiques, j'ai aussi pris l'initiative d'étudier les principes de l'acronyme S.O.L.I.D..

J'ai aussi beaucoup tâtonné pour obtenir le rayon et la position des disques qui épousent les cases du jeu sur l'image téléchargée pour les dimensions 900×800 avant d'appliquer la règle de trois pour les exprimer en fonction de la hauteur de l'image. Il a donc aussi fallu apprendre les rudiments de traitement des images pour implémenter la classe statique `ImageManager` afin de redimentionner l'image de la grille de jeu sans déformation via la règle de trois et les constantes géométriques obtenues (pour les dimensions 900×800).

Pour finir, pour que les coups joués dans la grille puissent s'enchaîner, qu'ils soient de la part des utilisateurs ou des *IA*, je n'ai pas tout de suite pensé à étendre la classe `Timer` en ajoutant un attribut de classe `PropertyChangeSupport` et en ré-implémentant la méthode `stop` pour en informer le contrôleur.

Pistes d'amélioration

Le respect des bonnes pratiques par cette implémentation facilitent la maintenabilité et l'évolutivité de ce programme.

L'intelligence artificielle peut être améliorée en affinant la fonction d'évaluation de la classe `AIPlayer`, en modifiant la méthode `wantedColumn` de cette dernière classe en récupérant la réponse d'un réseau de neurones, ou encore en développant des stratégies pour élaguer l'arbre en aval afin d'augmenter sa profondeur sans déclencher d'erreur de débordement de mémoire.

La mémoire peut aussi être économisée en simplifiant la classe `EvolutiveGridParser`. En effet, seul l'attribut `colorToArrayOfSetOfMonochromaticFourCellsInARow` est utilisé lors de l'évaluation des *IA*. Afin de diminuer la redondance de donnée décuplée par les arbres des *IA*, il serait donc pertinent de supprimer les attributs de la classe mère `ConstantGridParser` qui est classe fille de la classe `Grid` représentant simplement la grille de jeu, donc en définitive en supprimant la classe `ConstantGridParser` qui ne semble finalement pas avoir de raison d'être, en rendant la classe `EvolutiveGridParser` fille directe de `Grid`.

L'interface graphique peut également être changée ou améliorée sans effectuer de grands changements dans la vue et le contrôleur.

Liens étudiés

- section « Lalgorithme du min-MAX » de la présentation d'un projet d'informatique sur le site de la formation en informatique de l'université de Lille
- article « L'INTELLIGENCE ARTIFICIELLE DEFINITION - GENERALITES - HISTORIQUE - DOMAINES » sur la page de Dominique Pastre–Ollivier-Pallud, professeur d'informatique honoraire, sur normalesup.org
- discussion de forum « Création d'une classe générique » sur openclassrooms.com tutoriel pour implémenter une classe générique sur koor.fr
- tutoriel « Création interface graphique avec *Swing* les bases » sur waytolearnx.com
- tutoriel « Comment tracer des lignes, rectangles et cercles dans JFrame » sur waytolearnx.com
- tutoriel « Swinguez !! `JFrame`, `JPanel`, `JComponent`, `LayoutManager`, `BorderLayout` » sur codes-sources.commentcamarche.net
- tutoriel « Utilisation de stratégies de positionnement (layout managers) » sur koor.fr
- tutoriel « Introduction au dessin en Java » sur duj.developpez.com
- article de *Wikipédia* « Interface de programmation »
- tutoriel « Programmation Concurrente et Interfaces Interactives » de Nicolas Sabouret sur son site
- cours « Création interface graphique avec Swing : les bases » de Baptiste Wicht sur baptiste-wicht.developpez.com
- cours « Apprendre Java » d'Irène Charon sur perso.telecom-paristech.fr
- cours « Écrivez du code Java maintenable avec MVC et SOLID » de Lorraine Le Jan et de Glen Wolfram sur openclassrooms.com
- discussion de forum « SwingUtilities.invokeLater : à quoi ça sert ? » sur www.developpez.net/forums
- cours « Apprenez à programmer en Java » sur zestedesavoir.com
- discussion de forum « Exemples de jeu MVC » sur openclassrooms.com
- article *Wikipédia* « Event dispatching thread »
- « A Visual Guide to Layout Managers » de la documentation en ligne de *Java*
- tutoriel pour réaliser des graphiques sur remy-manu.no-ip.biz
- tutoriel « Comment redimensionner une image en Java » sur waytolearnx.com
- cours « Dessin *Java* et *Java 2D* » sur le site de Jean Berstel sur le site de l'université Gustave Eiffel et de l'Institut d'électronique et d'informatique Gaspard-Monge (IGM)
- thèses « Architecture des applications graphiques en *Java* » de Philippe Andary sur sur « HAL Normandie Université, archive ouverte institutionnelle de la communauté scientifique normande »
- section « Une conception sur le modèle MVC (Model-View-Controller) » du cours de *Java* d'Irène Charon sur sur perso.telecom-paristech.fr
- discussion de forum « Observer is deprecated in Java 9. What should we use instead of it ? » sur stackoverflow.com
- tutoriel « How to Write a Property Change Listener » de la documentation de *Java*
- chapitre « Les composants *Java beans* » www.jmdoudoux.fr
- discussion de forum « How to fix The constructor Timer(int, new ActionListener()) is undefined error in JAVA ? » sur stackoverflow.com

Instructions de compilation et d'exécution

- installer Java
- se positionner dans le répertoire du projet
- pour compiler le programme, soumettre au terminal la commande :

```
javac *.java $(find . -type f -name "*.java") -cp "../:*"
```

- pour exécuter le programme, soumettre au terminal la commande :

```
java Main
```