

Projet d'Architecture Logicielle
Projet Micro Éditeur de Figures Géométriques

BEDIN Dylan
TROTTIER Gaël

Chargé de TD : CÉLERIER Jean-Michaël
Enseignant de cours : AUBER David

Année universitaire 2016-2017

Table des matières

1	Introduction	2
2	Organisation du projet	3
3	Observer	4
4	Memento : undo/redo et Prototype	5
5	Sérialisation : sauvegarde et chargement des fichiers	6
6	Limites de notre programme	7

Chapitre 1

Introduction

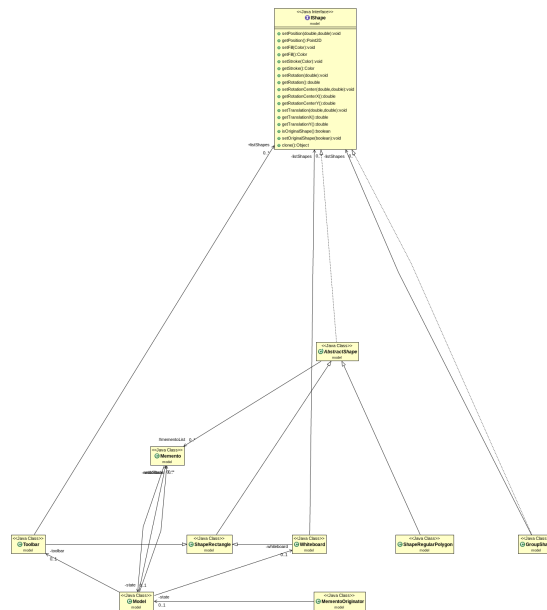
L'objectif de ce projet est d'obtenir un micro éditeur de figures géométriques. Les différentes étapes qui ont été implémentées sont les suivantes. Notre logiciel comporte deux formes, un rectangle bleu et un polygone bleu et est capable de sélectionner ces formes depuis la toolbar pour les positionner dans la whiteboard. De plus, nous avons mis en place un système d'undo/redo ainsi que rendu possible la sérialisation de la whiteboard. Nous n'avons pas implémenté la modification des objets par le biais des handlers ou l'association des groupes mais avons effectué un modèle de telle sorte que l'ajout de ces différentes fonctions soit aisé.

L'idée principale du projet est de faire en sorte que toutes les modifications visibles dans la View se fassent par le biais d'une mise à jour du Model. Dans ce rapport, nous allons détailler les différents design patterns utilisés et justifier les choix de ces derniers, ainsi que leur utilité dans le sein du projet.

Chapitre 2

Organisation du projet

Nous avons choisi de mettre en place un MVC (Model View Controller). On retrouve ainsi un package View, un package Model et un package Controller. Le Controller contient un objet “MouseEvents” présentant tous les événements souris réalisables ainsi qu’une classe “Controller” servant à lancer l’application et à instancier l’application. Cette dernière lancera donc la View et instanciera un Model. Nous avons fait le choix d’avoir un Model static disponible depuis la classe “Main”, lançant le programme et instanciant le Controller. Cela nous permettra d’y accéder aisément. Une première version présentait le Model comme étant un singleton, partant du principe que ce dernier était unique. Cependant, cette idée fut abandonnée lors de la mise en place du undo/redo à l’aide du design pattern Memento. Le Model se présente ainsi :



Chapitre 3

Observer

Le pattern Observer est au coeur du projet. C'est lui qui permet de faire en sorte que le Model mette à jour la View. Lorsque l'utilisateur effectue un évènement souris, cela fait appel à la classe "MouseEvents" située dans le package "Controller". Son rôle est de mettre à jour les éléments du Model. Ce dernier est une classe contenant la whiteboard et la toolbar du point de vue Model. Il s'agit d'un objet Observable. Ainsi, la View observe et override la méthode "public void update(Observable arg0, Object arg1)". Cette dernière saura en fonction de l'objet arg1 reçu la façon dont elle doit se mettre à jour. Nous avons décidé que les objets "toolbar" et "whiteboard" devaient être créés au lancement de la View. Ainsi, les premières méthodes appelées sont "createWhiteboard()" et "createToolbar()" dont les shapes présentes et la taille dépendent du modèle.

Une méthode importante de cette classe est "majShape(IShape shape)". En effet, quelque soit la modification effectuée sur une IShape, on appellera cette méthode afin que son équivalent en JavaFX soit update. On peut donc imaginer simplement que l'ajout d'une fenêtre permettant de modifier d'autres données instanciées à l'heure actuelle dans le Model mais pas dans la View, telles que la couleur ou le centre de rotation d'une Shape, se fasse aisément. Il suffira d'envoyer la shape à la méthode "update" qui appellera la même méthode "majShape" et mettra à jour toutes les données. Nous avons souhaité privilégier cette méthode à une autre visant à séparer chaque changement de type de donnée car, même si cette dernière est nécessairement plus lourde et longue à exécuter, elle offre une facilité d'utilisation et diminue grandement le code d'update, les Shape n'ayant pas à être différenciées selon le type de traitement souhaité.

Chapitre 4

Memento : undo/redo et Prototype

Le choix a été fait de mettre en place un Memento afin de pouvoir annuler et refaire des opérations. Pour cela, nous allons stocker des Model dans des piles d'objets Memento. Nous avons ainsi créé 2 classes. La classe Memento sert à stocker un Model et à pouvoir y accéder. La classe MementoOriginator permet de stocker un Model et s'occupera de sauvegarder ce dernier dans l'objet Memento. Enfin, la classe Model stocke 2 piles, une pile servant à effectuer des undo, l'autre servant à faire les redo. A l'instanciation, les deux sont vides. A chaque fois que l'on notifie les observateurs, nous faisons appel non pas directement à la méthode "notifyObservers" de la classe Observable mais à une méthode ayant un booléen "memento". Si celui-ci est set à "true", nous faisons appel à la méthode "saveMemento" qui sauvegarde le Model courant et l'empile dans la pile "undoStack". Ainsi, dès que nous voulons effectuer un undo, il suffit de dépiler cette pile. Or, afin de permettre de faire l'opération redo, il suffit, au moment du dépilement d'"undoStack", d'empiler dans la pile redo. On dépilera cette dernière afin de faire l'opération redo. Cependant, à chaque fois que l'on empile dans la pile "undo", l'idée est de vider la pile redo afin de ne pas obtenir des incohérences si l'on mêle des undo et des redo.

Afin de garantir que les Model stockés dans les piles de Memento sont des instances à part, non partagées, il aura fallu implémenter des méthodes clone sur les objets Model, mais aussi Whiteboard et Toolbar, que Model contient. Il aura donc fallu mettre en place le design pattern "prototype".

Chapitre 5

Sérialisation : sauvegarde et chargement des fichiers

Chapitre 6

Limites de notre programme