École Polytechnique de Montréal

Département Génie informatique et
Génie Logiciel

Cours LOG8430 - Travail Pratique N°1

Mise en Œuvre d'une Architecture Logicielle et Chargement Dynamique

Réalisé par:

Yan Xu 1754774

Olivier Pinon 1758047

Chunxia Zhang

Soumis à: Zéphyrin Soh et Yann-Gaël Guéhéneuc

Session : Hiver 2015

Guide du développeur

Introduction

Cette application permet d'appliquer une commande ou un ensemble de commandes sur un fichier ou un dossier.

Étape du travail

P1 - Architecture

Nous avons choisi une architecture de type MVC. Cela permet de bien séparer la vue, du modèle ainsi que des controleurs.

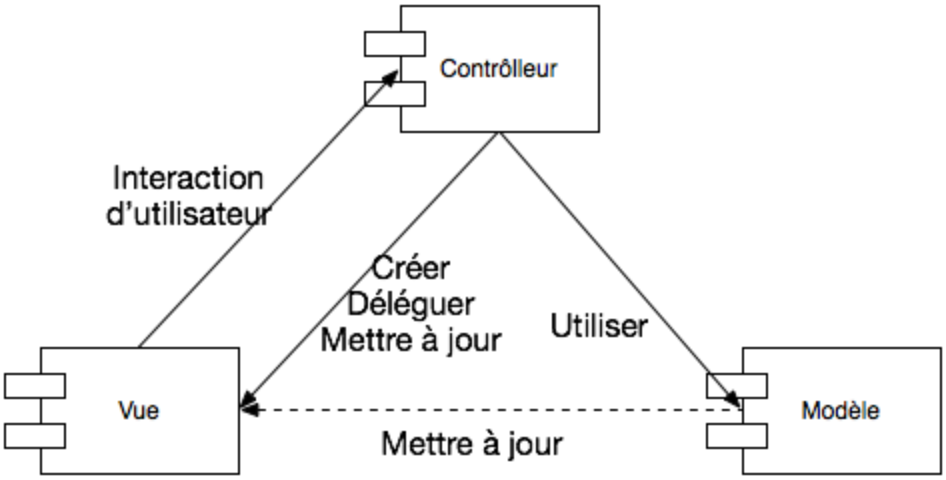

Figure 1. MVC architecture

Comme dans la figure 1, il y a trois types de module dans l'architecture MVC. Le modèle est la logique des fonction d'application. La vue est l'interface d'utilisation. La contrôlleur controlle la vue et utilise le modèle.

L'architecture est détaillée dans le diagramme UML suivant :

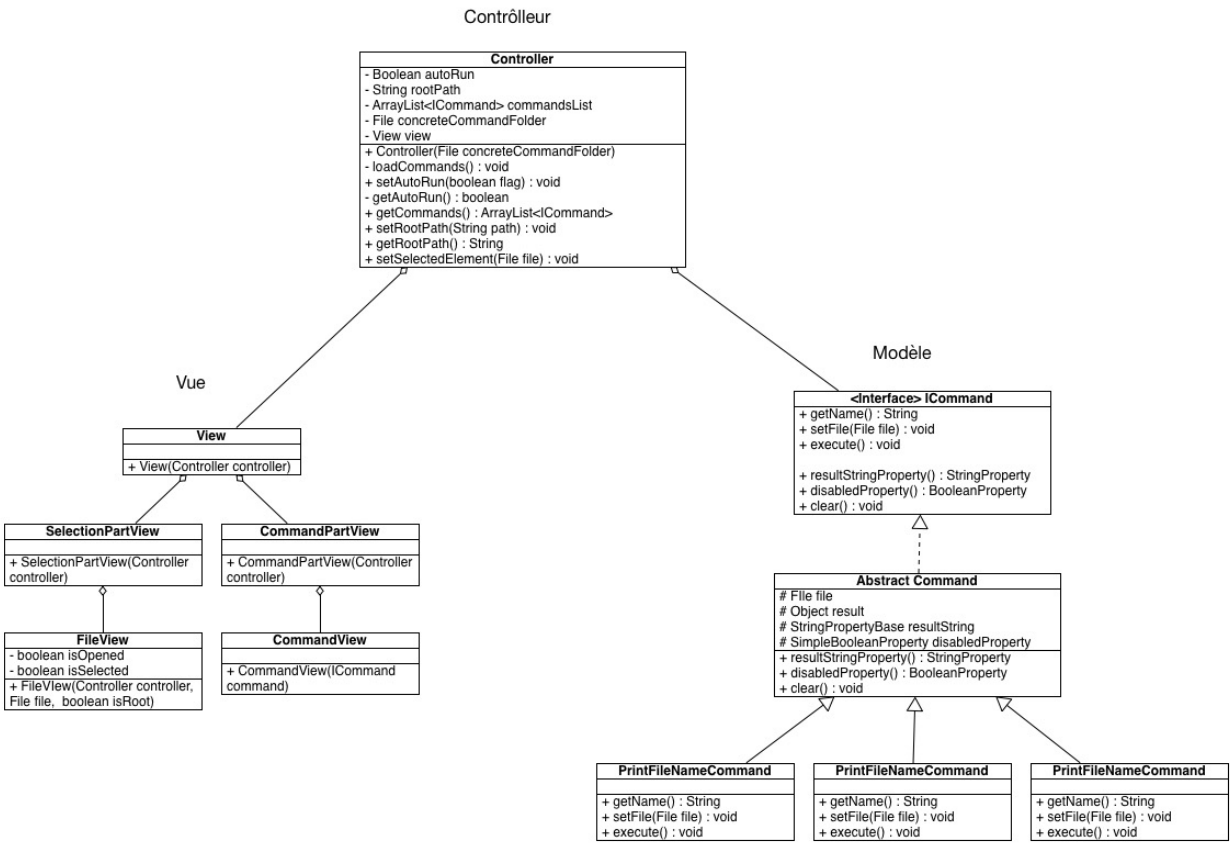

Figure 2. UML-classe diagramme d'application, la classe Main qui démarre le programme n'est pas incluse.

P1 - Patron de conception

Dans les différents composants, nous avons aussi utilisé divers patrons de conception :

- Nous avons implementé le patron controller "Command" pour les commandes. Les commandes ont une fonction setFile(File f) pour définir le "*Receiver"*, et une fonction execute() pour executer la commandes.

- Au début, nous avons implémenté le modèle et les vues en utilisant le patron de conception "Observer". En effet, le champ de texte de la vue observe le résultat de la commande, afin de se mettre à jour automatiquement (par exemple quand la commande est exécutée, ou que la commande est "cleared" ou que le fichier est changé). De même le bouton d'exécution des commandes observe le booléen "disableProperty" qui détermine si la commande est exécutable ou non. Finalement, nous avons utilisé des classes "Property" dans la bibliothèque de JavaFx pour réaliser le "data-bind".

- Nous avons pensé utiliser les patrons de conception “composite” et “visitor” pour construire l’arbre de fichiers et effectuer les commandes. Mais finalement, nous ne les avons pas implementées et nous ne sauvegardons qu’un rootPath dans le contrôlleur. L’arbre de fichier est donc construit dans la vue seulement car nous n’en avons pas besoin ailleurs.

P2 - Conception de l’interface des commandes

Nous avons défini l’interface *ICommand* implémentée par toutes les commandes :

ICommand

- `public void setFile(File f) : Définir le fichier sur lequel s’exécuter (en tant que “receiver” dans le patron de conception “Command”)`

- `public void execute() : Exécuter la commande, le résultat sera stocké dans un attribut donc il retourne “void”`

- `public void clear() : Remettre le resultat à vide`

- `public String getName(); // retourne le nom de la commande`
-
- `public StringProperty resultStringProperty() : Mettre à jour la vue du resultat`

- `public BooleanProperty disabledProperty() :  Désactiver/activer le bouton de la commande`

Cf programme:
Dans la pakage oxz.application.command:

- ICommand.java

P3 - Command vide et algorithme maître

Nous avons créé une classe “Controller”, qui a un attribut private ArrayList<ICommand> commandsList, pour stocker tous les commandes; Nous pouvons exécuter les commandes à partir de ce “Controller”.

On a aussi créé une abstract classe “Command”, pour mettre les méthodes communes, et 3 implémentations :

- PrintPathCommand : retourne le “filepath” du fichier
- PrintFileNameCommand : retourne le nom du fichier, et “error” si c’est un dossier
- PrintFolderNameCommand : retourne le nom du dossier, et “error” si c’est un fichier

P3 - Modification pour les différent receveurs des commandes

Le premier problème que la distinction fichier/dossier pose est le suivant : les boutons pour exécuter les commandes doivent être grisés si celles-ci ne sont pas compatibles avec le fichier sélectionné.
Pour cela nous adaptons la méthode `public void setFile(File file)` pour chaque commande et ajoutons une méthode `public BooleanProperty disabledProperty()`:

- Quand la méthode “`public void setFile(File file)`”, est appelée, on vérifie d’abord la compabilité de méthode avec l’élément sélectioné. S’ils ne sont pas compatibles, nous mettons l’attribut “`disabledProperty`” à Faux;

- public `BooleanProperty disabledProperty()`; Elle permet d’observer le booléen “`disabledProperty`” qui indique si oui ou non la commande est compatible. Les boutons des commandes dans l’IHM pourront donc observer cette variable.

P5) Junit Test pour l’algorithme “maître” et des commandes

Cf programme:
Dans la pakage oxz.application.test:

- ControllerTest.java
- PrintFileNameCommandTest.java
- PrintFolderNameCommandTest.java
- PrintPathCommandTest.java

Nous avons utilisé la reflection de Java pour tester les valeurs des attributs privés.

P6) Interface d’application

Cf programme:
Dans la pakage oxz.application.view:

- View.java
- CommandPartView.java
- SelectionPartView.java

- CommandView.java
- FileView.java

P7) Classe Command Concret

Cf programme:
Dans la pakage oxz.application.command.imp:
- PrintFileNameCommand.java
- PrintFolderNameCommand.java
- PrintPathCommand.java

P8) P9) Chargement Dynamique

Parce que nous stockons les commandes dans une `ArrayList<ICommand> commandsList`. Avant, nous ajoutons direction les commandes dans cette list. Pour réaliser cette fonction, nous avons écrit une methode `private void loadCommands()` du contrôleur permet de charger les classes et de mettre à jour ce ArrayList de commandes avec ces classes. Le patron “Iterator” peut s'appliquer ici, mais nous n’avons pas trouvé que c’est necessaire. De plus, Les commandes est le modèle dans nos architecture. Comme le contrôlleur est construit basé sur les modèle, au lieu de passer une liste de commandes à le contrôlleur, nous passons la repertoire des commandes à la contrôlleur.

De plus, nous avons implémenté la question bonus : pendant l’exécution du programme, si les fichiers *.class* sont changés, alors le programme se met à jour automatiquement. Pour cela, nous avons utilisé un *Thread* qui “écoute” les modifications du répertoire sur le disque; cette écoute est faite de manière passive, c’est à dire que le Thread attend un évènement du système lui indiquant que le répertoire a été modifié.

cf programme: oxz.application : Controlleur.java

P10) Test de Chargement Dynamique

Cf programme:
Dans la pakage oxz.application.test:
- ControllerTest.java

Nous avons utilisé la reflection de Java pour tester les valeurs des attributs privés.

P11) Interface usager pour le Chargement des Classes

Parce que nous avons implemté une chargeur automatique, donc nous n’avons pas besoin une interface pour faire ça.