École Polytechnique de Montréal

LOG8430
Architecture logicielle et conception avancée

TP1 : Mise en œuvre d'une architecture logicielle et chargement dynamique
Option 1 – Gestion de fichier
Guide du développeur

Soumis par
Julien Aymong (1629966)
Samuel Gaudreau (1631114)
Sylvester Vuong (1635535)

3 février 2016

Table des matières

P1 - Architecture choisi et patrons utilisés

Architecture

Après l'analyse des requis, nous avons constaté qu'une interface utilisateur devait être créée pour pouvoir exécuter les différentes commandes. Nous avons donc décidé d'utiliser le patron d'architecture logicielle Modèle-Vue-Contrôleur (MVC) pour réaliser cette partie des exigences. En effet, en plus d'être un patron que nous avons déjà eu l'occasion d'explorer et d'utiliser dans nos laboratoires à Polytechnique, il permet un contrôle sur ce que l'utilisateur peut exécuter ou modifier au niveau du modèle. Avec cette architecture, l'utilisateur interagit seulement avec la partie "Vue" du système. Cette vue affiche les attributs se trouvant dans le modèle. Nous utilisons le contrôleur pour modifier les éléments du modèle pour ensuite mettre à jour la vue. Cette architecture permet d'avoir une disparité entre l'information que voit l'utilisateur et les méthodes du système. De cette manière, il a seulement accès à ce qu'il à besoin. Ceci est important dans le cas présent, car l'utilisateur peut créer ses propres commandes ; il faut donc assurer une manipulation cohérente en tout temps.

Le modèle MVC a été appliqué comme suit : le Modèle représente l'arborescence de fichiers que l'utilisateur va choisir, la Vue est l'interface utilisateur (GUI) avec laquelle il va interagir et le Contrôleur est la logique interne permettant de mettre à jour le Modèle, charger et appeler les bonnes commandes.

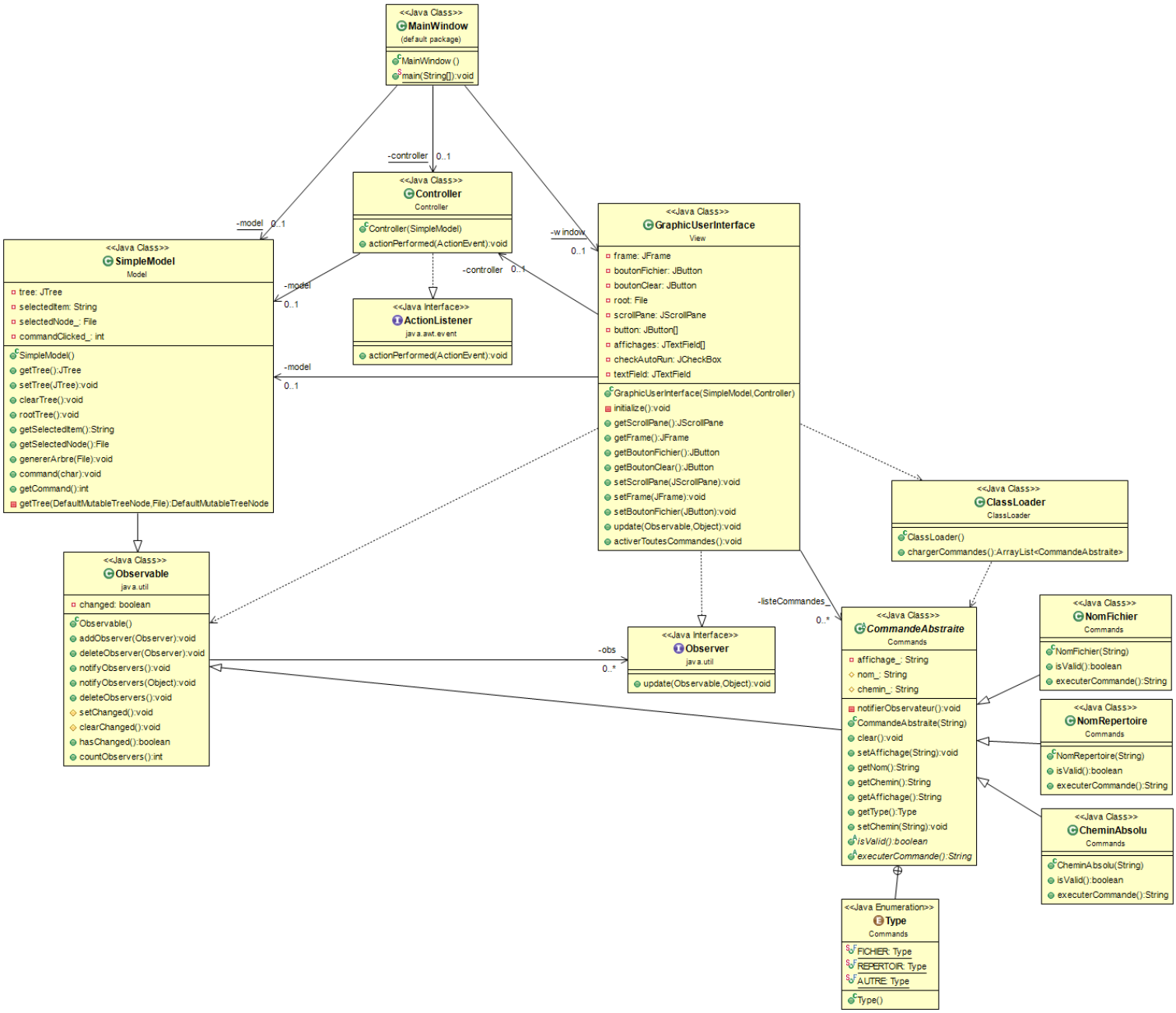
Patrons de conception

En plus du patron MVC, nous avons introduit d'autres patrons de conception dans notre architecture.

Le patron Observateur a été introduit à notre architecture MVC. Ainsi, en assignant la vue comme Observateur et le Modèle comme Observé, cela permet de faire la séparation entre ces eux parties, évitant des interactions directes entre la Vue (l'interface utilisateur) et le Modèle (l'arborescence de fichiers choisie). Dès que le modèle est mis à jour, la vue en est averti et se met à jour.

Pour les commandes, nous avons utilisé une interface commune avec le patron de conception Commande. Ainsi, lorsqu'un utilisateur décide d'implémenter ses propres fonctionnalités, il doit suivre un gabarit minimal pour assurer une cohérence en dérivant les nouvelles commandes du modèle de commandes abstraites.

Figure 1 – Diagramme de classe du projet (final)



P2 - Interface commune de commande

Nous avons décidé de rendre obligatoire l'implémentation de 2 fonctions dans notre interface commune des commandes.

La fonction `isValid()` permet de vérifier si la fonction est appelé sur le bon type d'élément dans l'arborescence, ainsi que si les valeurs ont bien été initialisé lors de la construction de la commande. Ainsi, une valeur booléenne est retournée par la fonction : `TRUE` si c'est le bon type d'élément, sinon `FALSE`.

La fonction `executerCommande()` est la fonction exécutant la logique même de la commande implémentée. Une fois le traitement terminé, le résultat est affiché dans la boîte de texte, à côté du bouton d'exécution de la commande. Ainsi, la valeur de retour de chaque commande est une `String`, soit le résultat de l'exécution.

P3 - Implémentation de commandes

Interface commune (commandes vides)

La classe abstraite permet de faire un modèle de base de commandes vides, d'interface commune. En déclarant les méthodes `isValid()` et `executerCommande()` comme étant abstraite, on oblige les personnes qui veulent écrire leurs propres commandes d'implémenter, au minimum, ces 2 fonctions.

```
public abstract class CommandeAbstraite extends Observable implements InterfaceCommande {
	private String affichage_;
	protected String nom_;
	protected String chemin_;

	private void notifierObservateur(){
		setChanged();
		notifyObservers();
	}

	public CommandeAbstraite(String chemin){
		chemin_ = chemin;
	}

	public void clear(){
		affichage_ = "";
		notifierObservateur();
	}

	public void setAffichage(String affichage){
		affichage_ = affichage;
		notifierObservateur();
	}

	public void setChemin(String chemin){
		chemin_ = chemin;
		clear();
	}

	public abstract boolean isValid();
	public abstract String executerCommande() throws Exception;
}
```

Algorithme « maître »

L'algorithme « maître » permet d'exécuter la commande associée au bouton, ou la commande appelé, le cas échéant.

```
public void actionPerformed(ActionEvent e){
	...
	else if("commandButton0".equals(e.getActionCommand()))
	{
		model.command1();
	}
	else if("commandButton1".equals(e.getActionCommand()))
	{
		model.command1();
	}
	else if("commandButton2".equals(e.getActionCommand()))
	{
		model.command1();
	}
}
```

P4 - Modification à l'interface de commande

Afin de faire la distinction entre les commandes s'appliquant seulement pour les dossiers ou les fichiers, un `ENUM` a été ajouté à la classe abstraite. Ainsi, la création d'une commande doit aussi s'appliquer sur un élément de type `FICHIER`, `REPERTOIR` ou `AUTRE`. Pour faire cette vérification, on vérifie le type de l'élément sélectionné lors de l'exécution d'une commande.

Les ajouts suivants ont donc été faits dans la classe abstraite :

```
public enum Type {
        FICHIER, REPERTOIR, AUTRE
}

...
protected Type getType(){
        File fichier = new File(chemin_);
        Type type;

        if(fichier.exists())
                type = Type.REPERTOIR;
        else if (fichier.isFile())
                type = Type.FICHIER;
        else
                type = Type.AUTRE;

        return type;
}
```

P5 - Tests unitaires JUnit sur l'algorithme « maître » et les commandes

La classe de tests unitaires `TestCommandes` permettent d'évaluer le bon fonctionnement de nos 3 commandes de bases, ainsi que sur la commande abstraite. Les tests évaluent si le bon type de l'élément sur lequel la commande est retournée, ainsi que seuls les commandes sur leur type appropriés s'exécutent sans problème.

P6 - Intégration des commandes à l'interface

Les boutons et les champs de texte pour les 3 commandes de bases ont été ajoutés à un `JPannel`, à droite de l'arborescence, pour respecter l'interface et les requis de la Figure 1 de l'énoncé du laboratoire. Ces derniers sont fixes.

P7 - Création de commandes

Voici nos implémentations des 3 commandes de bases, en suivant notre architecture implémentée. Aucun changement majeur n'a été effectué dans l'architecture après l'implémentation de réelles commandes.

Chaque commande nécessite au moins l'implémentation des fonctions `isValid()` et `executerCommande()` de notre classe abstraite.

Commande « Nom du fichier »

```
@Override
public boolean isValid(){
	if(chemin_ == null || chemin_.isEmpty())
		return false;

	if(getType() == Type.FICHIER)
		return true;

	return false;
}

@Override
public String executerCommande() throws Exception{
	if(!isValid())
		throw new Exception("Erreur, pas un fichier");

	int index = chemin_.lastIndexOf("\\");
	String cheminAffichage = chemin_;
	if(index != -1)
		cheminAffichage = chemin_.substring(index+1, chemin_.length());

	setAffichage("Nom du fichier: " + cheminAffichage);

	return getAffichage();
}
```

Commande « Nom du répertoire »

```
@Override
public boolean isValid() {
	if(chemin_ == null || chemin_.isEmpty())
		return false;

	if(getType() == Type.REPERTOIR)
		return true;

	return false;
}

@Override
public String executerCommande() throws Exception {
	if(!isValid())
		throw new Exception("Erreur, pas un repertoir");

	int index = chemin_.lastIndexOf("\\");
	String cheminAffichage = chemin_;
	if(index >= 0)
		cheminAffichage = chemin_.substring(index+1, chemin_.length());

	setAffichage("Nom du repertoir: " + cheminAffichage);

	return getAffichage();
}
```

Commande « Chemin absolu »

```
@Override
public boolean isValid() {
	if(chemin_ == null || chemin_.isEmpty())
		return false;
	return true;
}

@Override
public String executerCommande() throws Exception {
	if (!isValid()) {
		throw new Exception("Erreur, chemin null");
	}
	setAffichage(chemin_);

	return getAffichage();
}
```

P8 - Ajout de la fonctionnalité de commandes dynamiques

Afin de répondre à ces nouvelles exigences, nous avons ajouté une classe permettant de charger les commandes dans une liste, soit la classe `ClassLoader`. Une instance de cette dernière est créé dès l'initialisation de l'interface utilisateur et remplit sa liste des commandes trouvées. L'architecture déjà en place respectait bien cette contrainte, donc aucune modification majeure n'a dû être apporté au projet. Cependant, l'algorithme « maître » permettant de choisir la bonne commande à exécuter a été refactorisé pour choisir automatiquement la bonne commande selon le bouton appuyé dans la Vue.

Un patron Visiteur aurait pu être implémenté dans cet ajout à notre architecture. Cependant, nous avons jugé que cela n'était pas nécessaire. En effet, le fait d'avoir le patron Commande et l'interface commune limite le nombre d'actions possibles pour une nouvelle fonction. Une simple itération à travers les commandes chargées dans le programme est donc nécessaire.

P9 - Chargeur de classes

Le chargeur de classe a été construit en se basant sur un tutoriel trouvé sur Internet. Le chargeur commence par accéder au répertoire où sont stockés les commandes maisons (soit le répertoire (packages) « Commands ». Par la suite, il identifie dans ce dossier seulement les fichiers compilés (.class), puis vérifie si ce sont des instances de `CommandeAbstraite`. Ainsi, seulement les commandes correctement implémentées sont ajoutées à la liste de commande que le programme pourra exécuter.

P10 - Tests unitaires JUnit sur le chargeur de classes

Puisque l'énoncé spécifiait de réaliser les étapes P1 à P9, les tests unitaires pour le chargeur de classes n'ont malheureusement pas été implémentés.

P11 - Intégration du chargeur de classes

Le code a été modifié pour permettre de créer un nombre dynamique de boutons soient ajoutés dépendamment du nombre de commandes dans la liste de commandes abstraites, en mettant automatiquement à jour le texte sur ces derniers.