LOG4430 : Exercice sur les bibliothèques et le chargement de composantes dynamiques

Guide du développeur

Concepts :

Le programme a été conçue dans le but d'appliquer un ensemble de commandes sur un répertoire ou fichier spécifié en paramètre. Les commandes peuvent être appliquées indépendamment. Le programme intègre une interface usager simple pour appeler les commandes :

- un bouton permet de choisir un répertoire ou un fichier. Si un répertoire est choisi, le contenu du répertoire (et de ses sous-répertoires) est affiché dans une liste. Si un fichier est choisi, le nom du fichier est affiché seul dans la liste.
- un ensemble de commandes sont disponibles sur le côté du bouton et de la liste. Ces commandes permettent de réaliser diverses opérations sur le répertoire (et ses sous-répertoires et fichiers) ou le fichier choisi.

Les différents traitements a effectuer sur les répertoires et fichiers sont chargés dynamiquement, depuis un répertoire indiqué par l'utilisateur, au lancement du programme par l'intermédiaire d'un chargeur de classes (MyClassLoader.java). L'interface graphique et les boutons sont chargés en fonction du nombre de traitements (.class) présent dans le répertoire.

Architecture logicielle :

Notre logiciel utilise tout d'abord un patron commande. En effet, nous devons avoir la possibilité d'appeler n'importe quelle commande et de les exécuter les unes après les autres. Nous avons définit une interface de programmation commune à toutes les commandes (Commande.java). Ainsi, notre commande va communiquer une action à effectuer ainsi que ses arguments. La commande est envoyée à une méthode dans une classe qui va la traiter. En définissant, cette interface commune, nous aurons alors la possibilité de modifier les commandes très simplement en ne modifiant qu'une seule interface de programmation, puisque celle-ci est commune à toutes nos commandes. Il sera alors facile pour l'utilisateur de charger n'importe quelle commande dans l'application. De plus, il est très simple de

rajouter des commandes en implantant de nouvelle classes qui implémente de l'interface Commande sans à avoir à toucher au reste du code de l'application. Le processus de chargement de nouvelles commandes sera le suivant : l'usager du programme connaît le répertoire ou doivent se trouver les commandes. Il y met ses commandes. Au lancement du programme les classes des commandes sont chargées dynamiquement et apparaissent dans l'interface. Il a également la possibilité d'enlever des commandes puis d'en rajouter.

Notre logiciel est basé sur une architecture dont les composants sont:

Objet/données. L'accès à la base de données se fait par le biais des fichiers du dossier /répertoires.

Contrôle des données. Le contenu envoyé à l'utilisateur est vérifié par les fichiers du dossier racine.

Apparence. Affichage de l'interface utilisateur.

Il s'agit du principe de l'architecture MVC (Modèle-Vue-Contrôleur), en plus du patron commande qui reste à la base de l'application. La partie qui interagit avec l'utilisateur (Vue) est découplée des données qui caractérisent l'état de l'application (Modèle) par un Contrôleur qui gère l'exécution des commandes.

Une architecture MVC a de nombreux avantages :

- Les développeurs peuvent ajouter et corriger le code plus rapidement.
- N'importe quel designer peut travailler en tout e sécurité sur l'interface graphique de l'application sans avoir à comprendre ou même lire l'intégralité du code.
- Chaque composant peut être conçu indépendamment.
- La flexibilité, en effet si le client décide de changer l'interface utilisateur, nous aurons grâce à ce découpage, une grande maniabilité et facilité de changement.
- Notre application peut être testée indépendamment de son interface. Le nombre de canaux entre chaque composant est minimal.

C'est dans l'indépendance des trois couches cette architecture puise sa puissance et sa logique.

Il est ainsi facile de mettre à jour le design de son application, sans toucher aux données ni à la façon dont elles sont organisées. Inversement, il est aussi facile de modifier la structure de stockage des données ou la manière dont elles sont gérées sans bouleverser l'affichage.

Diagramme de classe du package Modèle

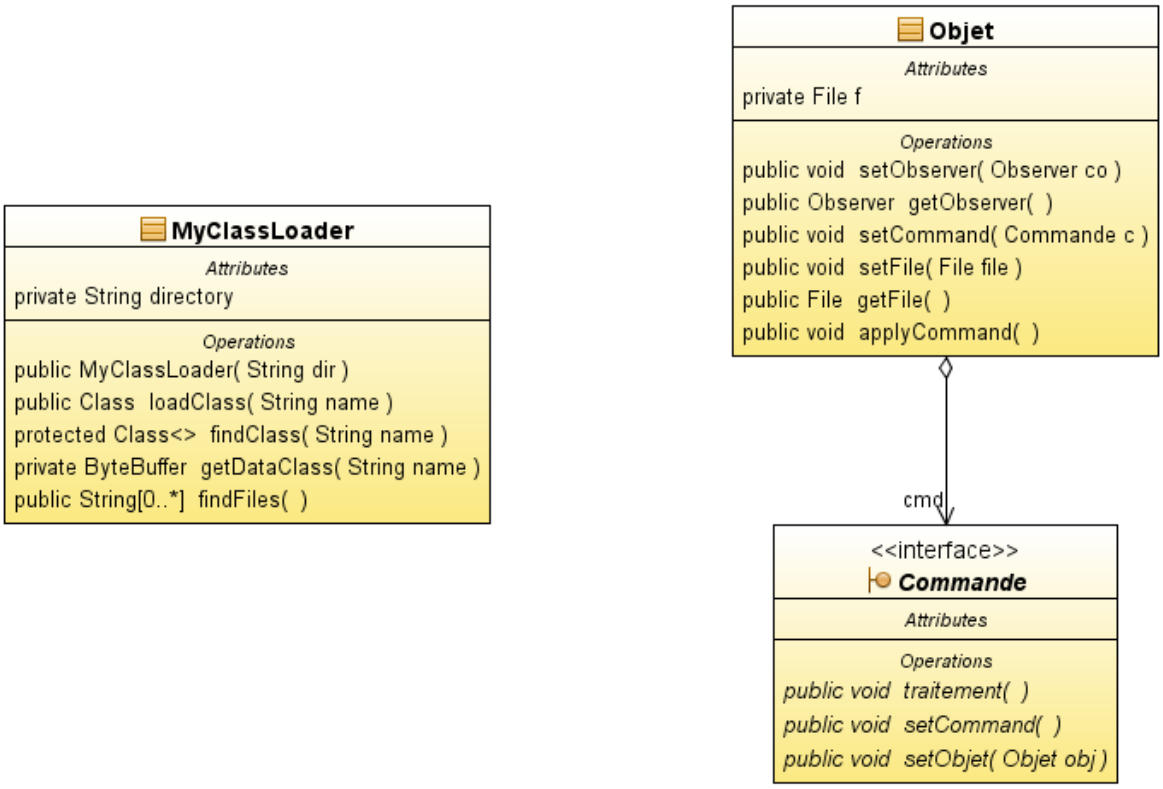


Diagramme de classe du package Vue

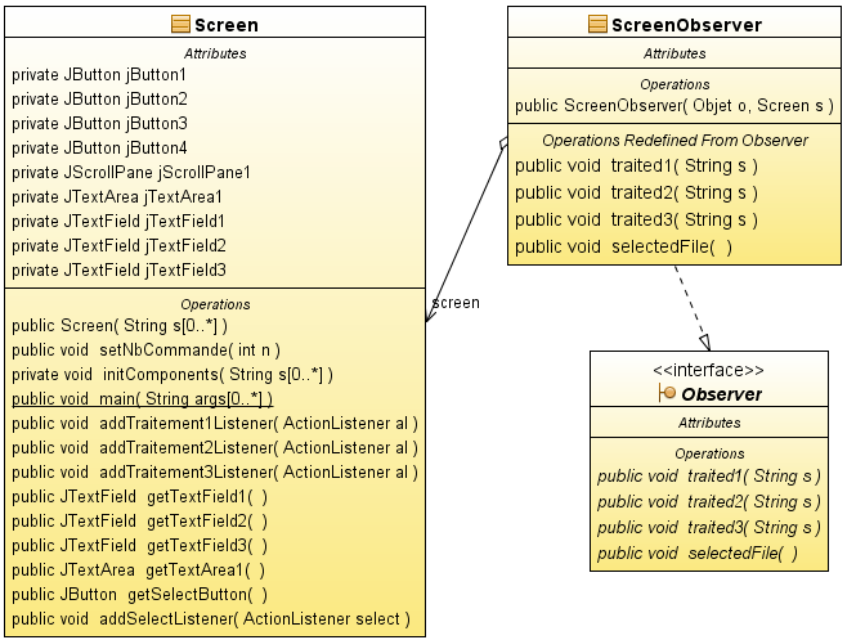
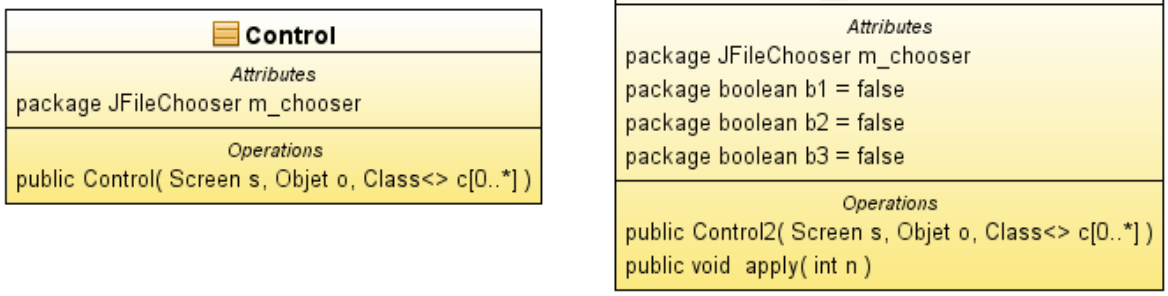


Diagramme de classe du package Contrôleur



Au sein de notre application nous faisons également appel au patron observateur.

Ce patron de conception explicite la relation entre un objet dit Observable et un mécanisme observant les changements d'état de celui appelé Observateur . Ainsi, dans notre application, le modèle ne contient aucun lien direct vers le contrôleur ou la vue, il comporte la liste de nos fichiers/répertoires (base de données) ainsi que leur propriétés et la liste des méthodes pour les commandes qui seront exécutées par le contrôleur. Sa communication avec la vue s'effectue au travers de ce patron. La vue (interface utilisateur) est un "observateur" du modèle qui est lui "observable". Dans notre système on souhaite que lorsque l'utilisateur clique sur un bouton commande, la vue affiche le résultat de la commande ou encore affiche le nom d'un fichier ou le contenu d'un répertoire. Ainsi lorsque que l'utilisateur clique sur un bouton, le modèle de donnée change et notifie ses observateurs et l'interface est mise à jour. Nous avons intégré ce patron à notre application car il permet de lier de façon dynamique un observable à des observateurs. Cette solution est faiblement couplée ce qui lui permet d'évoluer facilement avec le modèle.

tests :

Pour tester le programme, nous avons implanté partiellement un premier traitement qui doit pouvoir affiché la taille d'un fichier. Celui-ci étant implanté juste pour tester le bon fonctionnement de l'application, elle ne marche que sur les fichiers, mais montre une architecture MVC fonctionnelle et implantable.

Un test Junit, testClassLoader.java, a été codé pour tester les méthodes de la classe MyClasseLoader redéfinissant classLoader.