

LOG8430 : TP2

Mise en œuvre d’une architecture logicielle et chargement dynamique

Alexandre Chenieux - Thomas Neyraut - Alexandre Pereira

Polytechnique montréal

Table des matières

[Introduction 2](#_Toc445740050)

[Architecture du logiciel 3](#_Toc445740051)

[a) Cas d’utilisation 3](#_Toc445740052)

[b) Processus du logiciel 4](#_Toc445740053)

[c) Diagramme de classes 5](#_Toc445740054)

[d) Diagramme de paquetage 6](#_Toc445740055)

[e) Design pattern 7](#_Toc445740056)

# Introduction

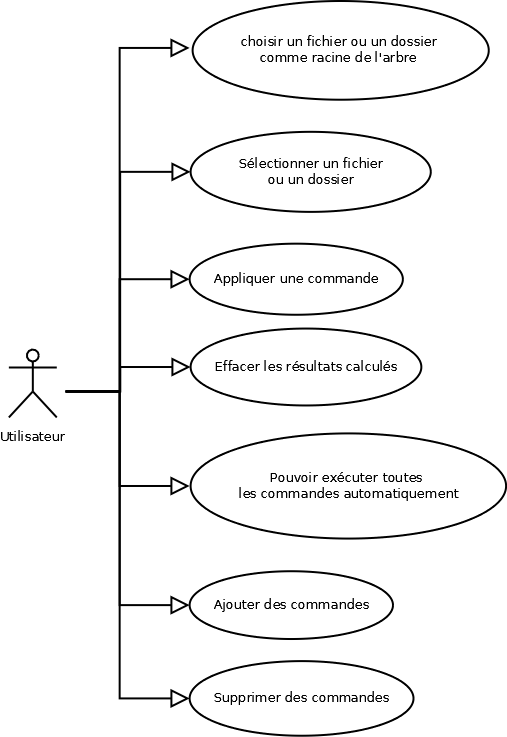
Ce document présente l’architecture du logiciel que nous avons implémenté afin de répondre à notre demande client. Il nous a été demandé de concevoir un programme permettant d’appliquer un ensemble de commandes sur des fichiers ou des dossiers spécifiés par l’utilisateur. De plus, le programme doit afficher une fenêtre graphique permettant son utilisation. Pour des soucis d’organisation et de développement, nous avons utilisé un dépôt GitHub pour faciliter le développement collaboratif du logiciel (<https://github.com/PolymtlAC/LOG8430-TP2>). La partie ci-dessous présente l’architecture du logiciel, son fonctionnement et ses fonctionnalités.

# Architecture du logiciel

Dans cette partie, un ensemble de diagrammes et de figures présentent l’architecture, le fonctionnement et les fonctionnalités du logiciel.

## Cas d’utilisation

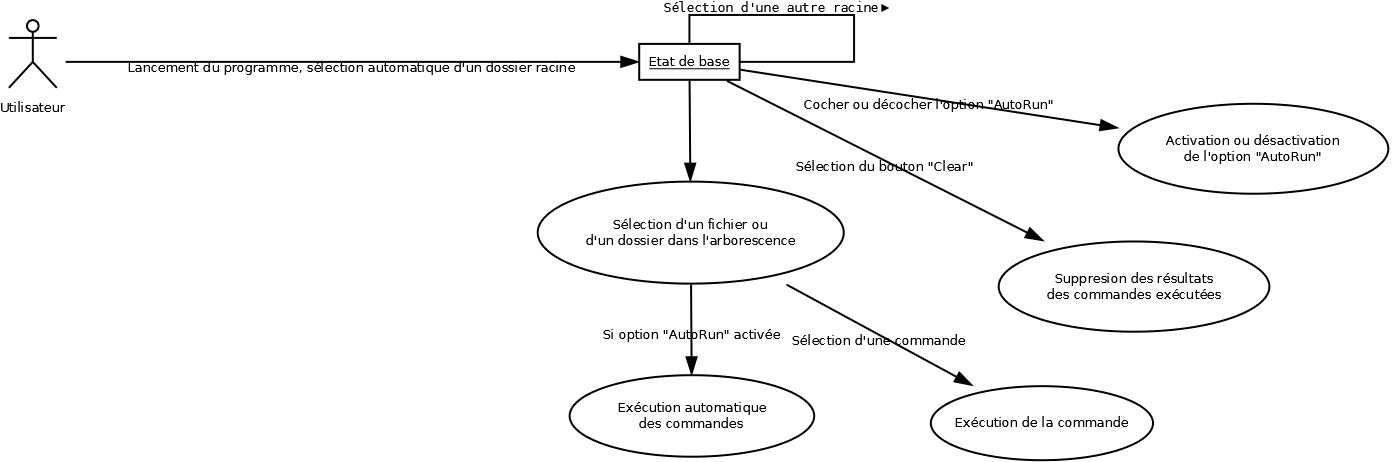
Dans un premier temps, nous avons listé les acteurs et les différentes fonctionnalités (actions effectuables par un acteur) que devrait comporter notre logiciel. En effet, il est important dans un processus de développement de commencer par lister les fonctionnalités afin d’être certains de ne rien oublier, et de faire valider cette liste par le client. Nous avons réalisé le diagramme UML des cas d’utilisation suivant :



**Figure 1. Diagramme de cas d’utilisation**

## Processus du logiciel

Une fois après avoir réfléchi sur les différentes fonctionnalités que notre logiciel devra intégrer, nous avons réfléchi sur les suites d’actions pouvant s’offrir à un utilisateur au cours de son expérience. L’objectif ici est de définir si certaines actions peuvent être effectuées uniquement dans certains contextes (par exemple : après une ou plusieurs actions), et si certaines actions doivent être réalisées avant que l’utilisateur puisse utiliser le logiciel librement. La **Figure 2** ci-dessous présente les différentes actions pouvant être effectuées par l’utilisateur. Au lancement du logiciel, ce-dernier va sélectionner automatiquement un dossier racine pour l’arborescence. Ce dossier est par défaut le dossier « Racine » de l’utilisateur. Par la suite, l’utilisateur va pouvoir effectuer d’autres actions de son choix en sélectionnant un fichier ou un dossier dans l’arborescence avant de pouvoir exécuter une commande.

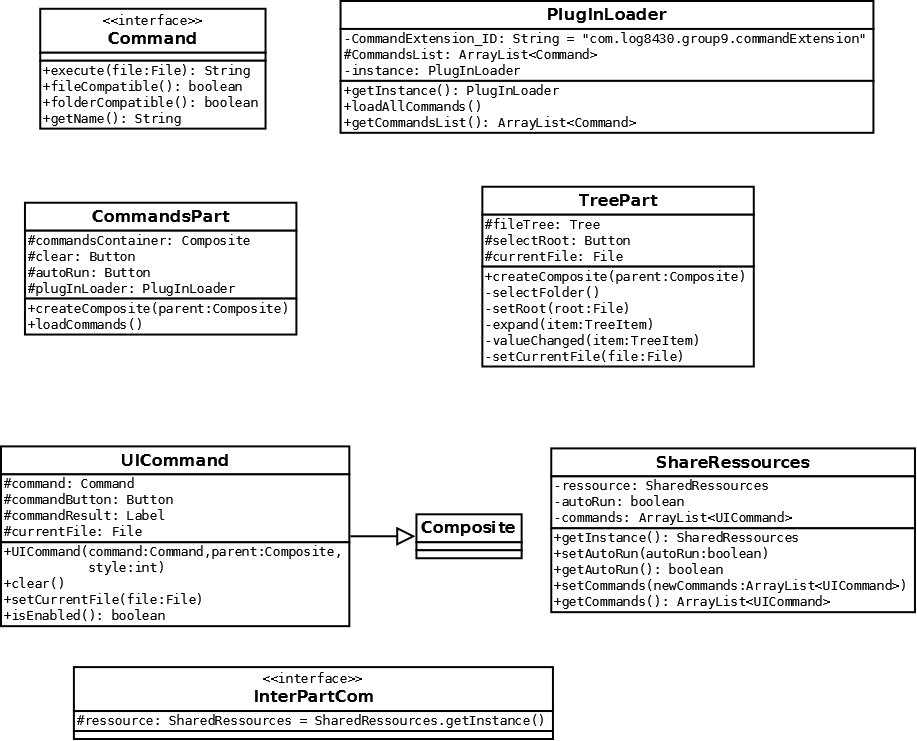


**Figure 2. Processus d’utilisation du logiciel**

## Diagramme de classes

Avant de commencer l’implémentation de notre logiciel, nous avions commencé à réfléchir sur sa structure à l’aide d’un diagramme de classes. Au cours de l’implémentation, nous avons complété notre diagramme de classes que vous trouverez ci-dessous (**figure 3**). Comme on peut le remarquer sur le diagramme, le programme comporte 5 classes :

* CommandsPart : Cette classe permet de définir l’interface graphique (éléments graphiques et actionListener) liée à une commande. Cette interface est composée d’un bouton permettant d’exécuter la commande et d’un champ texte permettant d’afficher le résultat de la commande.
* TreePart : Cette classe permet de définir l’interface graphique (éléments graphiques et actionListener) représentant l’arborescence des dossiers et fichiers, ainsi que les boutons « AutoRun » et « Clear ».
* UICommand : Cette classe permet de définir le regroupement des interfaces graphiques liées aux commandes (instances de la classe CommandsPart).
* Command : Cette interface définit les méthodes présentes dans les commandes issues d’autres plugins.
* PlugInLoader : Cette classe permet de charger les plugins des commandes.

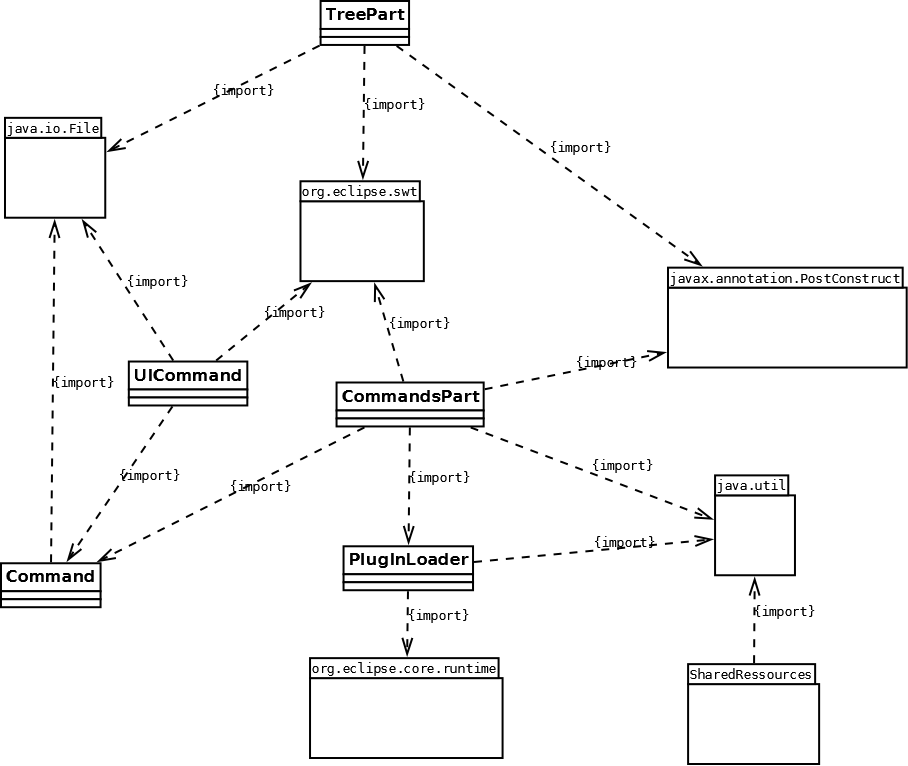


**Figure 3. Diagramme de classes**

Chacune des commandes présentes dans le logiciel sont implémentées séparément dans des plugins différents tout en respectant l’interface Command. Pour ajouter une commande, il faut l’implémenter sous la forme d’un plugin en respectant l’interface Command. Ensuite, il faut ajouter le plugin (.jar) dans le dossier « plugins » d’Eclipse et inclure dans les fichiers de configuration du projet ce nouveau plugin.

## Diagramme de paquetage

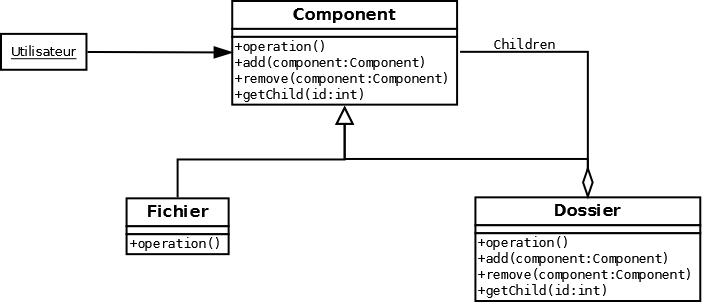
Afin de présenter sommairement les différentes librairies Java que nous avons utilisées dans nos différentes classes, nous avons réalisé un diagramme de paquetage (**figure 4**).



**Figure 4. Diagramme de paquetage**

## Design pattern

Comme le montre le diagramme de paquetage, nous avons utilisé les librairies org.eclipse.swt.widgets.Tree et java.io.File afin de représenter le système de dossiers et de fichiers dans notre logiciel. Ces deux librairies se basent sur le design pattern Composite dont voici dans notre cas le schéma (**Figure 6**).



**Figure 6. Schéma du design pattern Composite**

L’idée de ce design pattern est de représenter deux types d’entités (fichier et dossier) ayant la même nature (Component) et pouvant subir des opérations (dans notre cas des commandes). L’une de ces types entités (dossier) peut contenir des entités enfants (dossier ou fichier). Ce design pattern est très utile quand l’on souhaite établir des structures arborescentes entre des objets et les traiter uniformément. L’application de ce design pattern nous permet :

* D’avoir une hiérarchie de classes dans laquelle l’ajout de nouveaux composants est simple,
* De simplifier l’utilisation, l’utilisateur n’a pas à se préoccuper de l’objet accédé.

Nous avons aussi implémenté un singleton pour la classe PlugInLoader.