

LOG8430 : TP1

Mise en œuvre d’une architecture logicielle et chargement dynamique

Alexandre Chenieux - Thomas Neyraut - Alexandre Pereira

Polytechnique montréal

Table des matières

[Introduction 2](#_Toc441682115)

[I. Architecture du logiciel 2](#_Toc441682116)

[a) Cas d’utilisation 2](#_Toc441682117)

[b) Processus du logiciel 3](#_Toc441682118)

[c) Diagramme de classes 4](#_Toc441682119)

[d) Diagramme de paquetage 6](#_Toc441682120)

[e) Le design pattern Observer/Listener 6](#_Toc441682121)

[f) Le design pattern Composite 7](#_Toc441682122)

[II. Interface de programmation des commandes 8](#_Toc441682123)

[III. Modification de l’architecture du logiciel suite aux nouvelles demandes du client 9](#_Toc441682124)

[IV. Test JUnit 11](#_Toc441682125)

[Conclusion 11](#_Toc441682126)

# 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-TP1>).

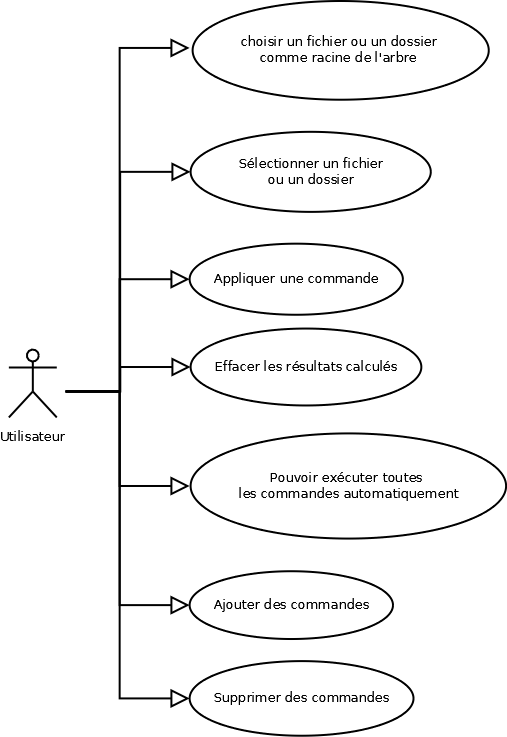
La première partie de ce document présente l’architecture du logiciel, son fonctionnement et ses fonctionnalités. La seconde partie présente plus en détails les aspects techniques permettant la programmation des commandes. Enfin, dans une dernière partie, nous présentons l’ensemble des modifications de notre architecture et de notre code que nous avons dû effectuées suite aux nouvelles demandes client.

# 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

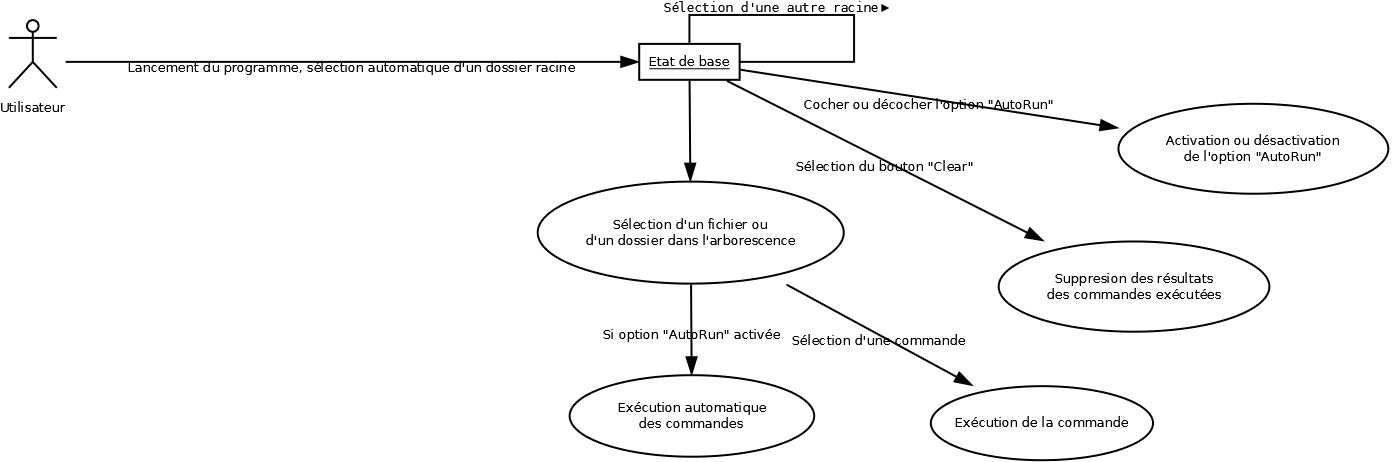
Dans un premier temps, nous avons listé les acteurs et les différentes fonctionnalités (actions effectuables par un acteur) que devrait comporter notre logiciel. En effet, il est important dans un processus de développement de commencer par lister les fonctionnalités afin d’être certains de ne rien oublier, 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 « Home » 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.

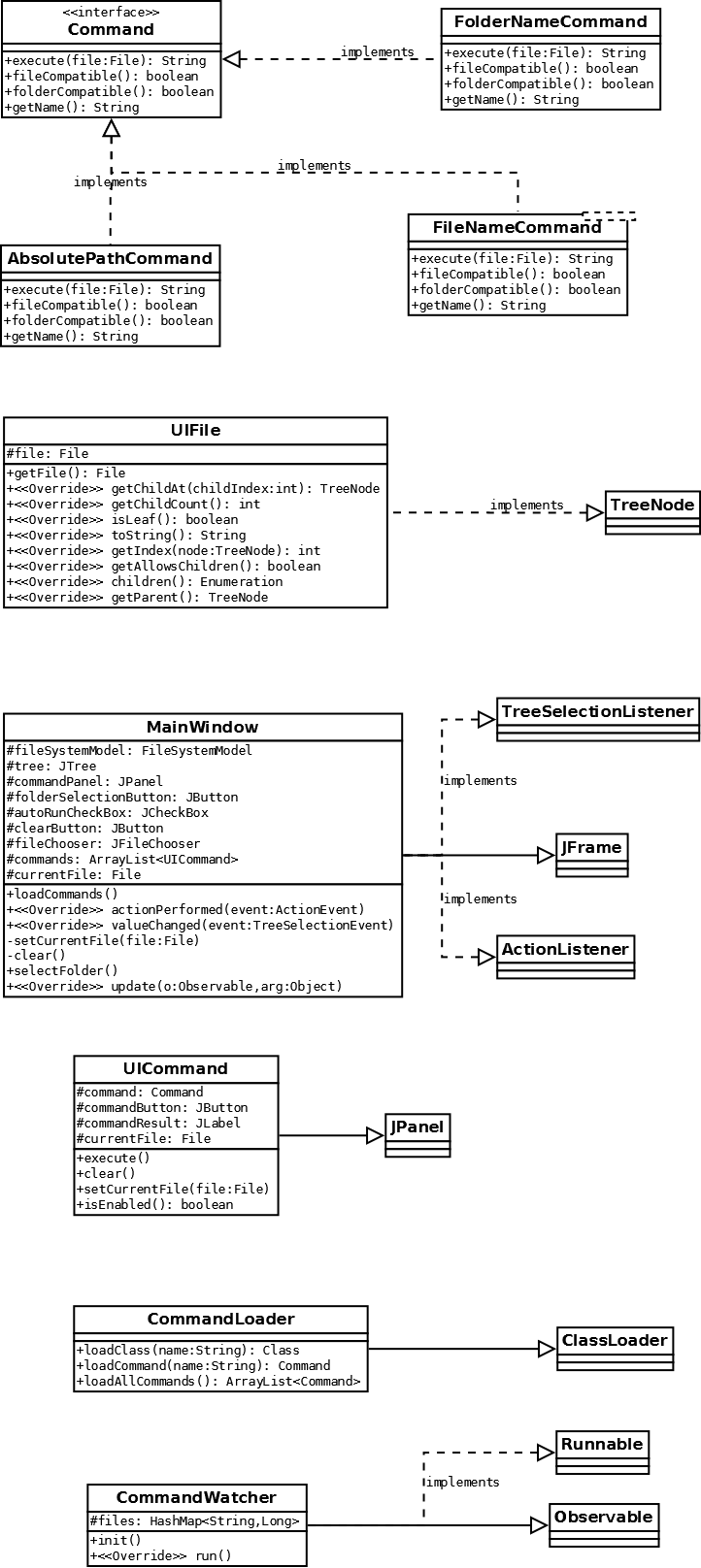


**Figure 2. Processus d’utilisation du logiciel**

## 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 6 classes :

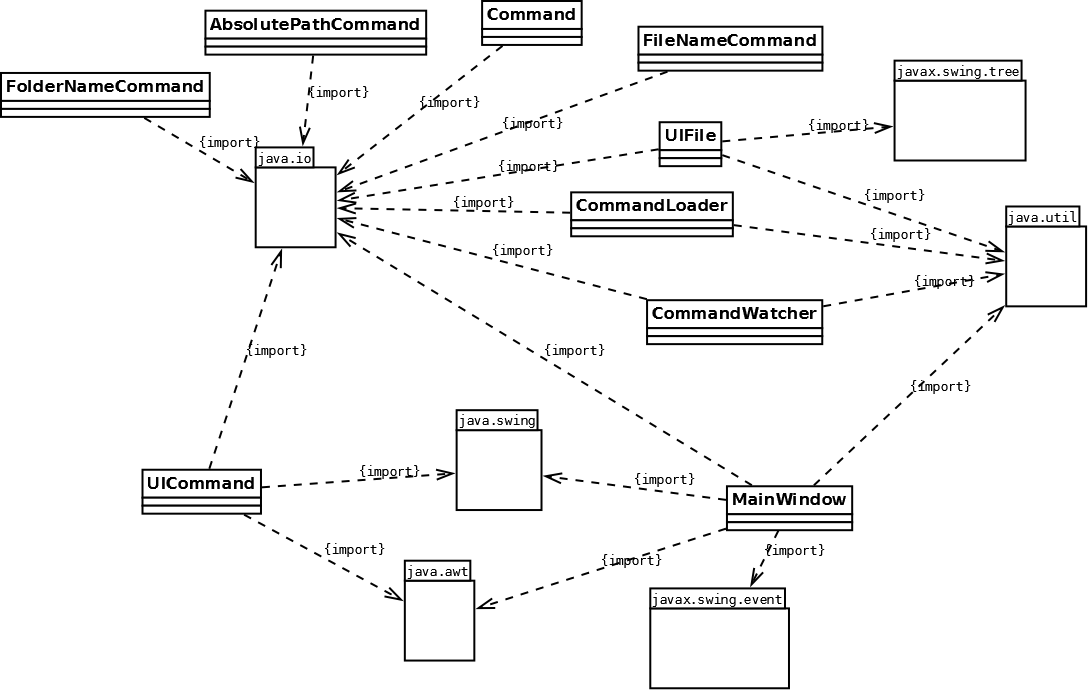
* Command : Cette classe est une classe interface permettant de définir les commandes applicables à un dossier ou/et à un fichier,
* FolderNameCommand : Cette classe implémente la classe Command et permet de définir les commandes applicables à un dossier,
* FileNameCommand : Cette classe implémente la classe Command et permet de définir les commandes applicables à un fichier,
* AbsolutePathCommand : Cette classe implémente la classe Command et permet de définir les commandes applicables à un dossier et à un fichier,
* UIFile : Cette classe permet de représenter l’arborescence des dossiers et fichiers,
* MainWindow : Cette classe permet de définir et d’afficher la fenêtre graphique principale du logiciel, elle permet aussi de gérer certains évènements provenant de cette fenêtre,
* UICommand : Cette classe permet de définir l’élément graphique complet permettant à l’utilisateur de lancer une commande en particulier parmi celles proposées, et permettant d’afficher les résultats des différentes commandes.



**Figure 3. Diagramme de classes**

## Diagramme de paquetage

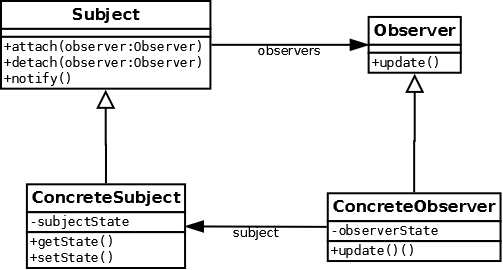
Afin de présenter sommairement les différentes librairies Java que nous avons utilisées dans nos différentes classes, nous avons réalisé un diagramme de paquetage (**figure 4**).



**Figure 4. Diagramme de paquetage**

## Le design pattern Observer/Listener

Comme explicité ci-dessus, nous avons implémenté une interface graphique et pour cela nous avons utilisé le design pattern Observer/Listener, dont voici le schéma générique (**figure 5**).



**Figure 5. Schéma du design pattern Observer/Listener**

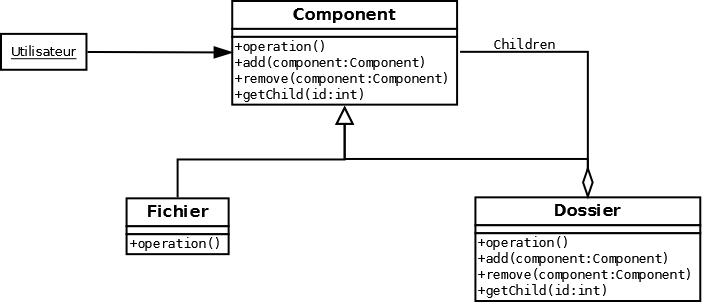
Ce design pattern est très utile pour la gestion d’évènements. Etant donné que Java nous fournit déjà un pattern Observer, nous l’avons utilisé (via les « Listener » et « Observable » / « Observer »).

Nous avons utilisé ce design pattern pour afficher les résultats des commandes exécutées. En effet, l’interface graphique propose une liste de commandes à exécuter. Une première solution était de laisser chaque UICommand observer elle-même son bouton d’action pour exécuter la commande en cas de clic. Nous avons préféré implémenter une seconde solution dans laquelle il y a une liste de UICommand dans la fenêtre principale (MainWindow) qui contient l’algorithme maître ainsi il est plus facile et efficace d’implémenter la fonctionnalité « AutoRun » (puisqu’il suffit de l’ajouter dans la boucle parcourant les commandes).

Le second cas d’utilisation du pattern Observer est pour la fonctionnalité permettant d’ajouter/modifier/supprimer des commandes pendant l’exécution du programme. Le CommandWatcher est un « Observable » et l’algorithme maître est une « Observer » et change son interface lorsque le watcher a vu une modification.

## Le design pattern Composite

Comme le montre le diagramme de paquetage, nous avons utilisé les librairies java.swing.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 :

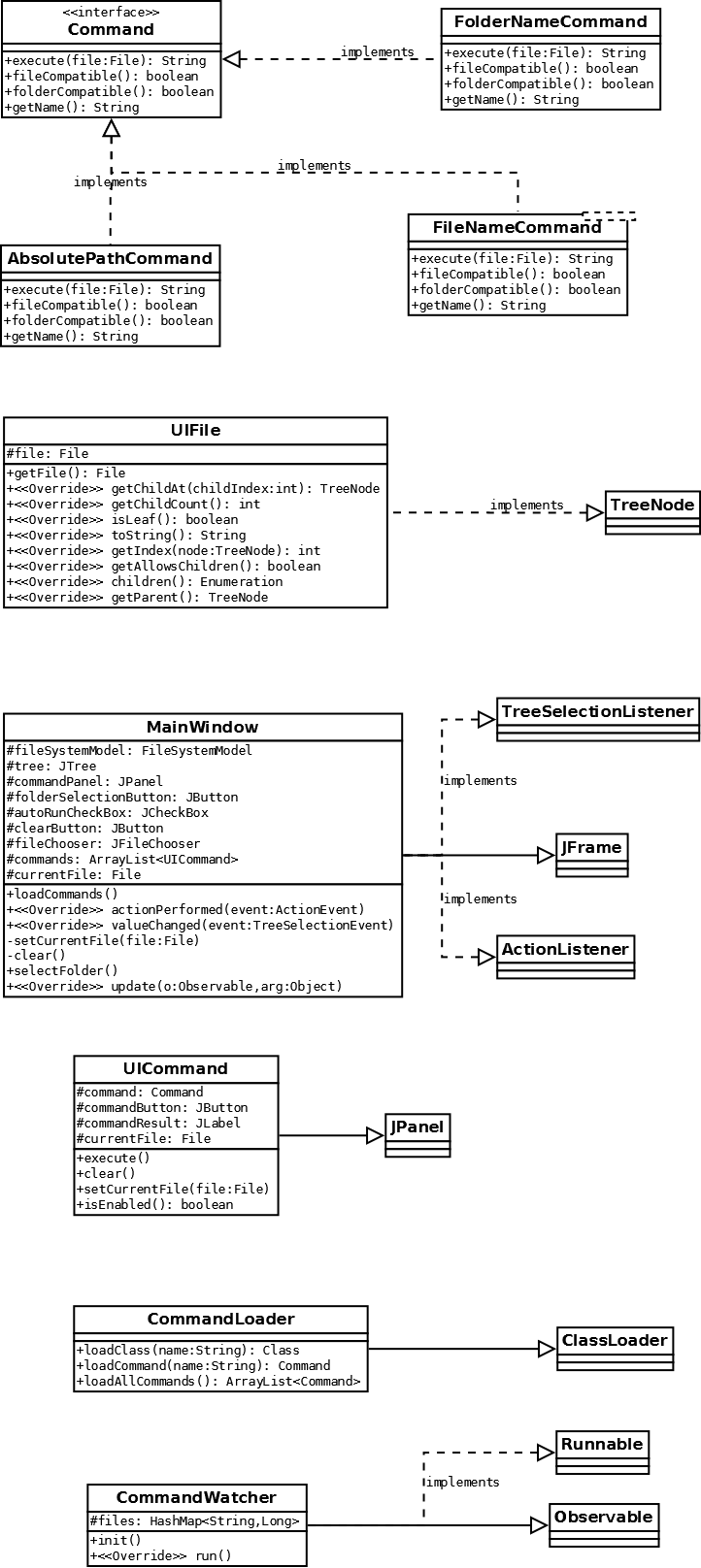
* D’avoir une hiérarchie de classes dans laquelle l’ajout de nouveaux composants est simple,
* De simplifier l’utilisation, l’utilisateur n’a pas à se préoccuper de l’objet accédé.

# Interface de programmation des commandes

Afin d’intégrer l’ensemble des commandes proposées par notre logiciel, nous avons choisi d’implémenter l’interface Command, comme présenté dans l’extrait du diagramme de classes ci-dessous (**Figure 7**). Par la suite, il fallait faire la distinction des commandes exécutables pour des fichiers uniquement, avec celles exécutables pour des dossiers uniquement, et enfin avec celles exécutables quelques soit la nature de l’objet (fichier ou dossier). De plus, il fallait qu’il soit facile pour le programmeur utilisant notre interface de spécifier sur quel type d’objet s’applique la commande. Pour cela, on s’est inspiré de la façon dont Java gère JTree et notamment l’interface TreeNode : en ajoutant des méthodes dans notre interface Command.

La fonction « execute », qui permet d’exécuter la commande et qui renvoie les résultats sous la forme d’une chaine de caractères,

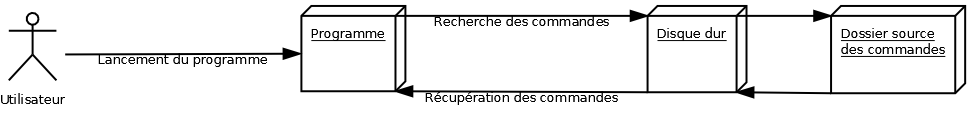
* La fonction fileCompatible, qui renvoie un boolean explicitant si la commande est compatible pour un fichier,
* La fonction folderCompatible, qui renvoie un boolean explicitant si la commande est compatible pour un dossier.



**Figure 7. Extrait du diagramme de classes**

# Modification de l’architecture du logiciel suite aux nouvelles demandes du client

Les demandes du client ont par la suite évolué. Il nous a été demandé de charger dynamiquement les commandes au lieu que celles-ci soient codées en dur dans le programme. Pour répondre à cette nouvelle exigence, nous avons dû un peu modifier l’architecture de notre logiciel. L’interface Command n’a pas changé, seule la façon dont on instancie les commandes dans l’algorithme maître a changé.

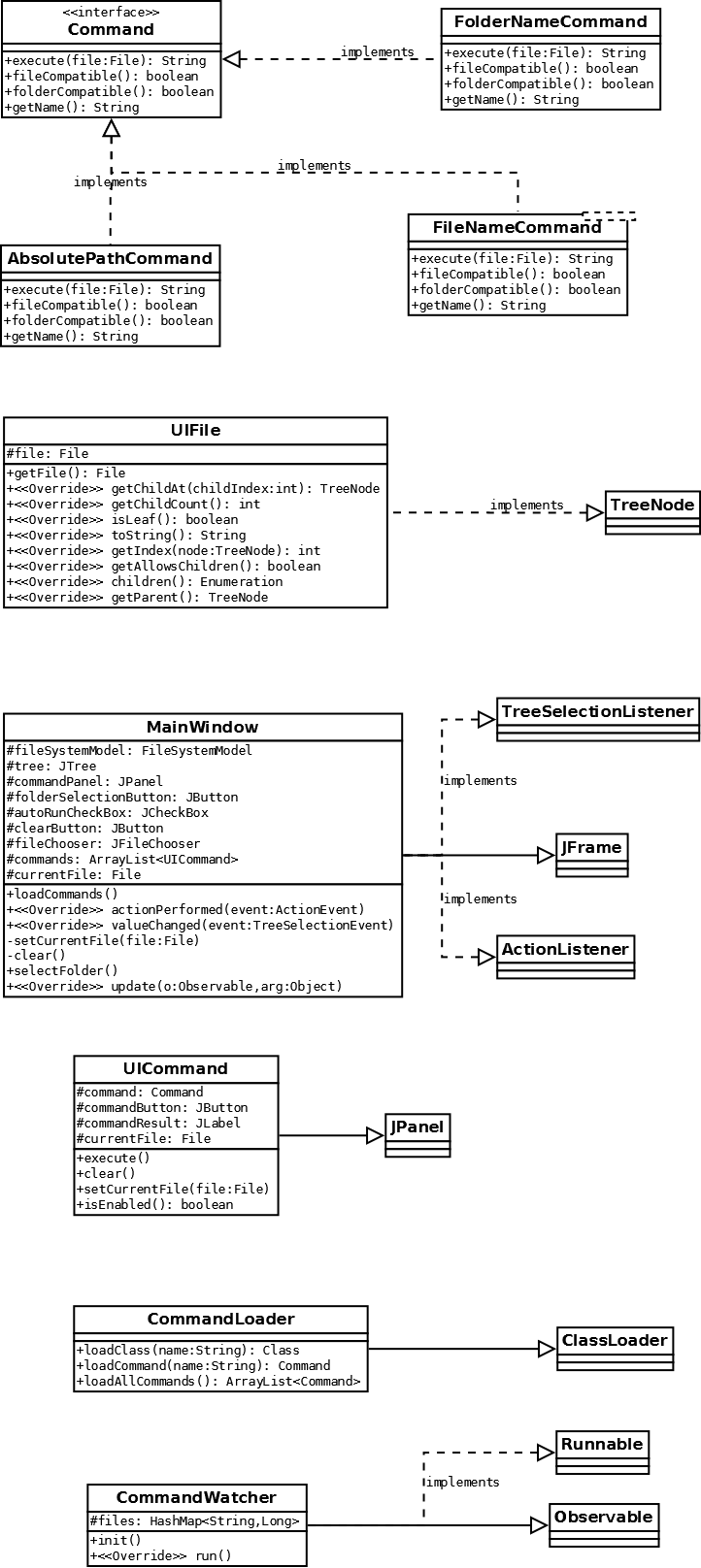
Au lancement du logiciel, ce-dernier va récupérer sur le disque dur dans un dossier spécifique l’ensemble des commandes applicables aux dossiers et fichiers par le logiciel. Le diagramme de déploiement suivant (**Figure 8**) représente ce processus effectué par le logiciel pour chercher les commandes.

**Figure 8. Diagramme de déploiement, recherche des commandes sur le disque dur**

Afin de charger dynamiquement au lancement du logiciel les commandes présentes dans un dossier du disque dur, nous avons implémenté deux nouvelles classes :

* La classe CommandLoader permet comme son nom l’indique de charger les commandes (fichiers en .class présents dans le dossier « commands »),
* La classe CommandWatcher sert d’observer le dossier contenant les commandes et d’en informer l’algorithme maître lorsqu’il y a une modification.

Suite à ces modifications, nous obtenons le diagramme de classe (**Figure 9**) suivant :



**Figure 9. Diagramme de classe modifié**

Par la suite, nous avons notamment implémenté de nouvelles commandes afin de vérifier cette partie de la maintenabilité le logiciel. En effet, nous avons pu tester qu’il était très simple de concevoir de nouvelles commandes en utilisant l’interface « Command » implémenté. Nous avons ajouté les commandes suites :

* FolderNumberOfChildCommand : cette classe implémente une commande permettant de connaitre le nombre de fichiers et de dossiers présents dans le dossier sélectionné,
* OpenCommand : cette classe implémente une commande permettant d’ouvrir le fichier ou le dossier sélectionné,
* ParentCommand : cette classe implémente une commande permettant de connaitre le dossier parent du fichier ou du dossier sélectionné.

# Test JUnit

Tout au long du développement, nous avons implémenté quelques tests afin de vérifier le bon fonctionnement d’une partie du logiciel. Le fichier CommandTest.java comprend un ensemble de tests permettant de vérifier le bon fonctionnement de la totalité des commandes implémentées. Dans le cadre de ces tests, un dossier « dossierTest » a été ajouté contenant un fichier « fichierTest ». Le fichier CommandTest.java comporte aussi un test permettant de vérifier le bon fonctionnement de classe CommandLoader. Pour finir, certaines méthodes étant un peu plus complexe à tester, les autres tests effectués ont consisté à exécuter le logiciel pour tester directement ses fonctionnalités et son bon fonctionnement.

# Conclusion

En conclusion, nous nous sommes appuyés sur plusieurs design pattern (composite, observer/listener) dans la conception de l’architecture de notre logiciel. Nous avons aussi implémenté un ensemble de tests nous garantissant le bon fonctionnement d’une partie du programme. Enfin, nous avons pu tester qu’il était simple d’ajouter de nouvelles commandes à notre logiciel.