UNIVERSITÉ DE RENNES 1

istic Informatique Électronique

Rapport de ACO

Analyse et Conception à Objets

Réalisé par : Amadou DIA et Romaric KOUAME

Enseignant : M. Mickaël FOURSOV

Master 1 MIAGE, Groupe 2D

Décembre 2020

Introduction

La conception et le développement d'applications sont au cœur de la formation du miagiste. C'est ainsi que le module ACO fait partie des différentes unités d'enseignement en Master 1. L'objectif dudit module est de permettre aux étudiants d'analyser et de concevoir des applications de qualité. Des patrons de conception aux tests unitaires, tout ce que l'on a besoin pour concevoir une application utilisable à grande échelle est enseigné durant ce module.

C'est dans ce cadre, qu'on était chargé de développer un mini éditeur de texte en console de type vim ou en interface graphique de type notepad ++. L'objectif de ce projet étant de mettre en pratique tous les enseignements théoriques et pratiques vus en cours.

I. Les fonctionnalités de l'application

Les fonctionnalités basiques de l'éditeur sont :

- Le texte à éditer est contenu dans un buffer.
- À l'intérieur de ce texte, l'utilisateur peut déterminer une sélection (avec un début et une fin).
- Tout texte saisi par l'utilisateur vient remplacer le contenu de la sélection dans le buffer.
- L'utilisateur peut copier le contenu de la sélection dans le buffer dans un presse-papier clipboard
- Le contenu de la sélection peut aussi être copié dans le presse-papier puis supprimé (CUT)
- L'utilisateur peut coller le contenu du presse-papier à la place du contenu de la sélection dans le buffer.

Au delà de ses fonctionnalités basiques, l'utilisateur doit pouvoir enregistrer ses actions et les rejouer mais aussi défaire ou refaire des actions sans limitation sur la longueur de l'historique (l'utilisateur peut ramener l'éditeur dans son état initial)

II. Les outils et technologies utilisés

Ce projet est entièrement codé en Java. Entre autres technologies et outils utilisés, nous avons :

- ➢ **Draw.io** : Draw.io est un éditeur de diagrammes UML en ligne. Cet outil nous a été très utile pour la réalisation des différents diagrammes de classes et de séquences.
- ➢ **IntelliJ** : IntelliJ est un IDE (environnement de développement) très puissant développé par JetBrains. Au-delà de la complétion de code qu'il fournit, il dispose d'outils puissants qui permettent d'accélérer le développement.
- ➢ **Gitlab** : Ce projet étant collaboratif, nous avions besoin d'un outil qui faciliterait le travail en binôme.

- **JUnit** : JUnit est un framework de tests unitaires pour le langage Java.
- **Mockito** : Framework pour effectuer des tests unitaires en Java

III. Réalisation de l'application

La réalisation de l'application s'est déroulée en trois étapes. Dans un premier temps le développement d'une version 1 qui implémente les fonctionnalités basiques de l'éditeur, puis une version 2 qui est une amélioration de la version 1 avec de nouvelles fonctionnalités et enfin une version 3 qui est l'éditeur complet avec toutes les fonctionnalités demandées.

L'application est constituée de deux différentes parties : le moteur et l'interface utilisateur. Le moteur est composé de l'API (Engine, Selection), de l'implémentation et des tests. L'Engine récupère les actions de l'utilisateur et les exécute sur le buffer, Selection nous permet de sélectionner du texte du buffer. L'interface utilisateur n'étant pas l'objet du projet est juste une console basique qui permet d'exécuter les actions sur le moteur.

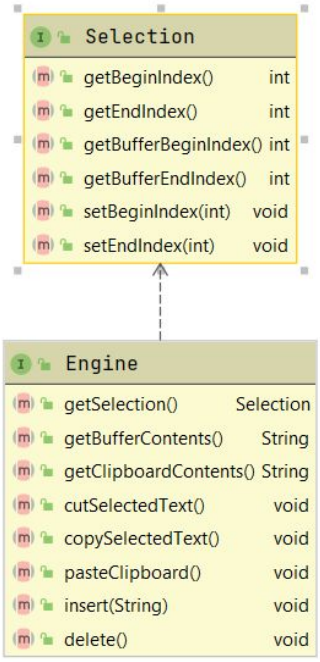


Diagramme de classe de Engine et Selection

A. Version 1

Dans cette version nous avons effectué l'implémentation de Engine et de Selection. Pour découpler les actions de l'utilisateur et le traitement, nous avons utilisé le patron de conception **Command**. Ainsi chaque action de l'utilisateur est associée à une commande. Exemple : l'insertion de texte est représentée par une commande **InsertCommand** qui se charge de réaliser l'action insertion de contenu dans le buffer. Pour la mise en œuvre de ce pattern, nous avons une interface Command et les actions de l'utilisateur sont représentées par des commandes concrètes.

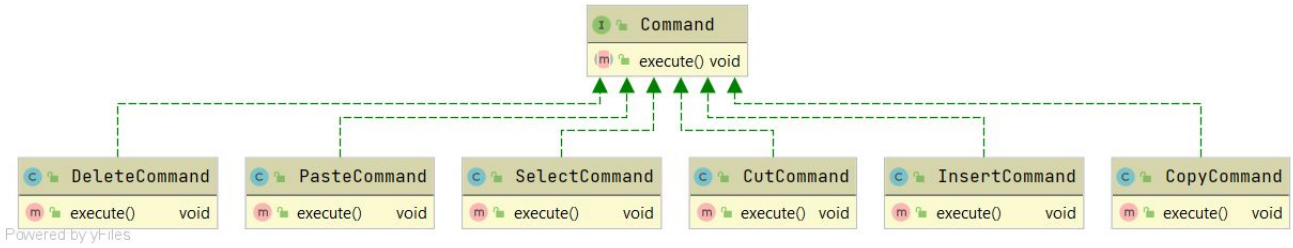


Diagramme de classe de Command et des commandes concrètes

Les étapes d'exécution d'une insertion de texte sont visibles dans le diagramme de séquence ci-dessous.

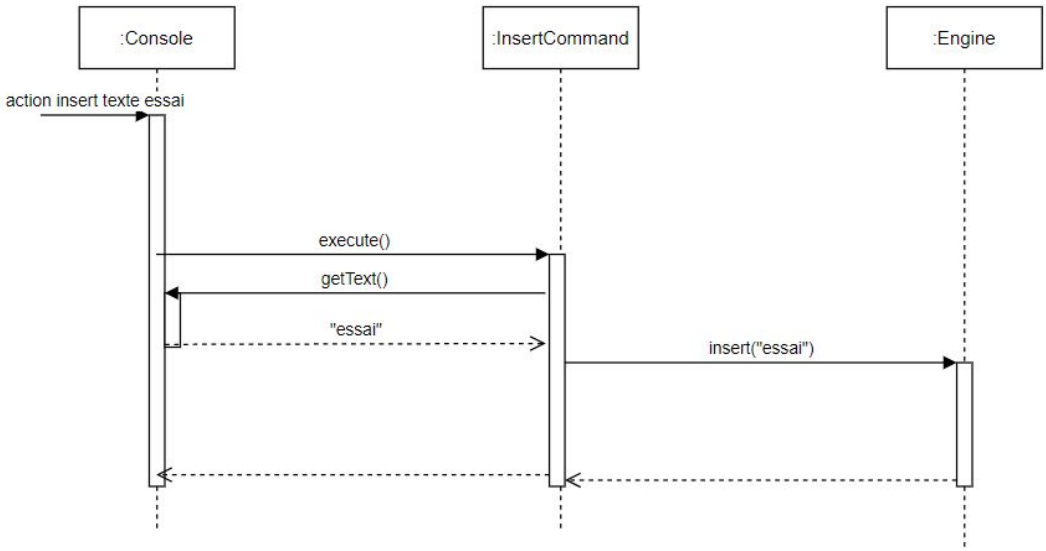


Diagramme de séquence d'insertion de texte

L'action Select fonctionne de la même manière que Insert. L'utilisateur tape la commande Select puis récupère les index à partir de l'invoker (Console)

Les autres actions (Paste, Copy, Delete, Cut) ont un fonctionnement beaucoup plus basique. L'utilisateur tape la commande correspondante puis elle est exécutée et l'action correspondante est réalisée sur le buffer.

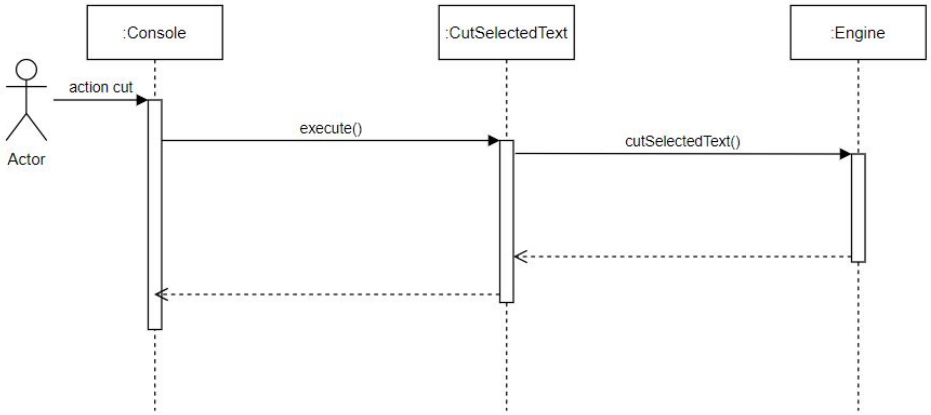


Diagramme de séquence de l'action couper

Le diagramme ci-dessous montre l'interaction des commandes avec l'Engine. Par souci de clarté, nous n'avons pas représenté toutes les commandes concrètes sur le diagramme.

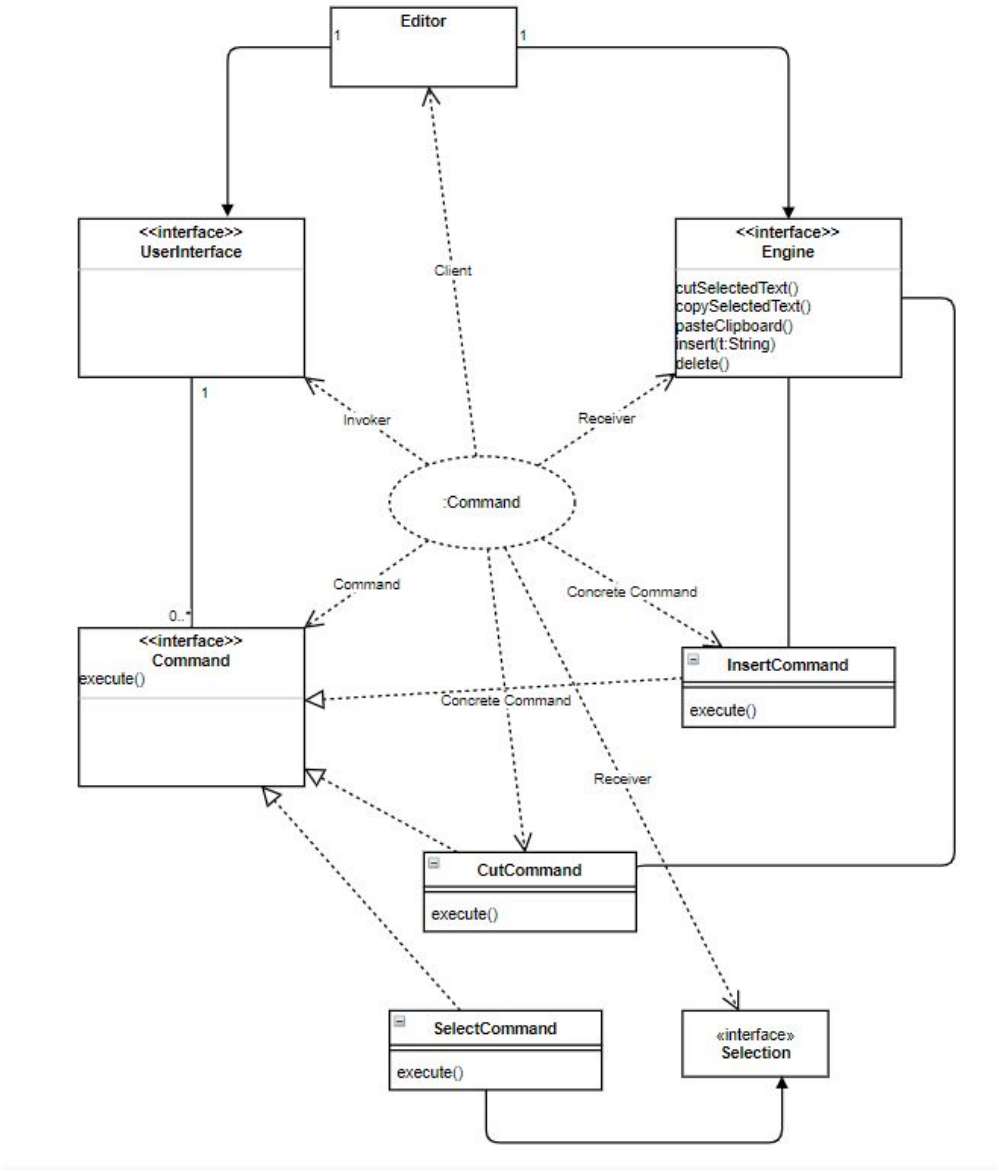
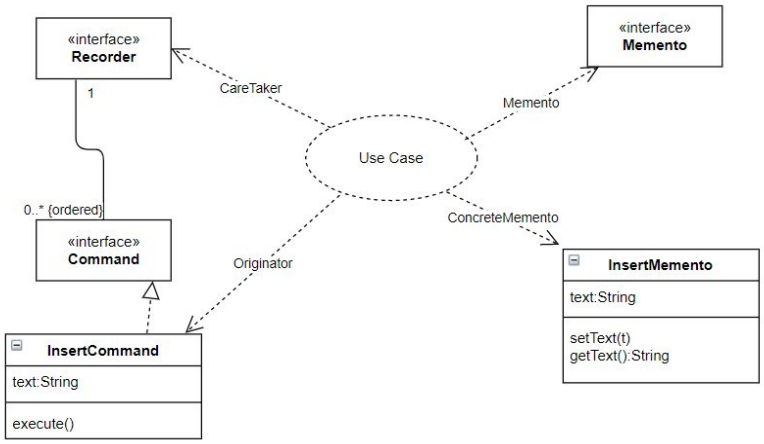


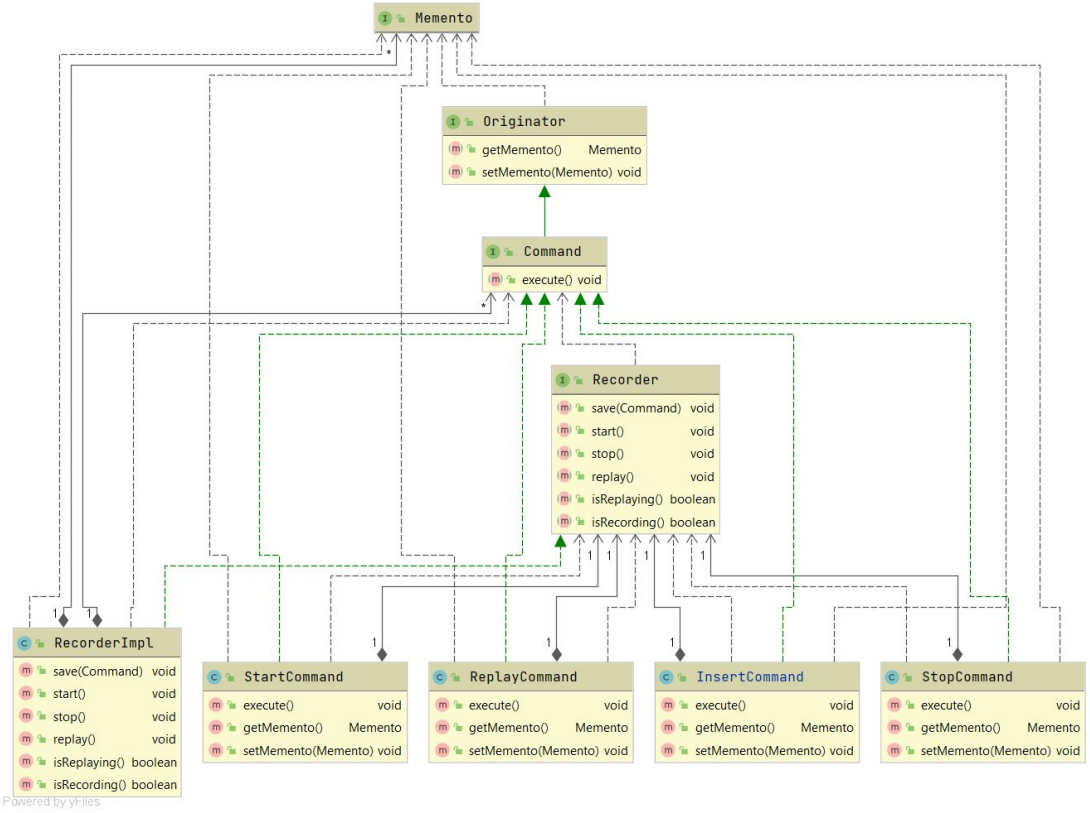
Diagramme de classe de l'Engine et des commandes

B. Version 2

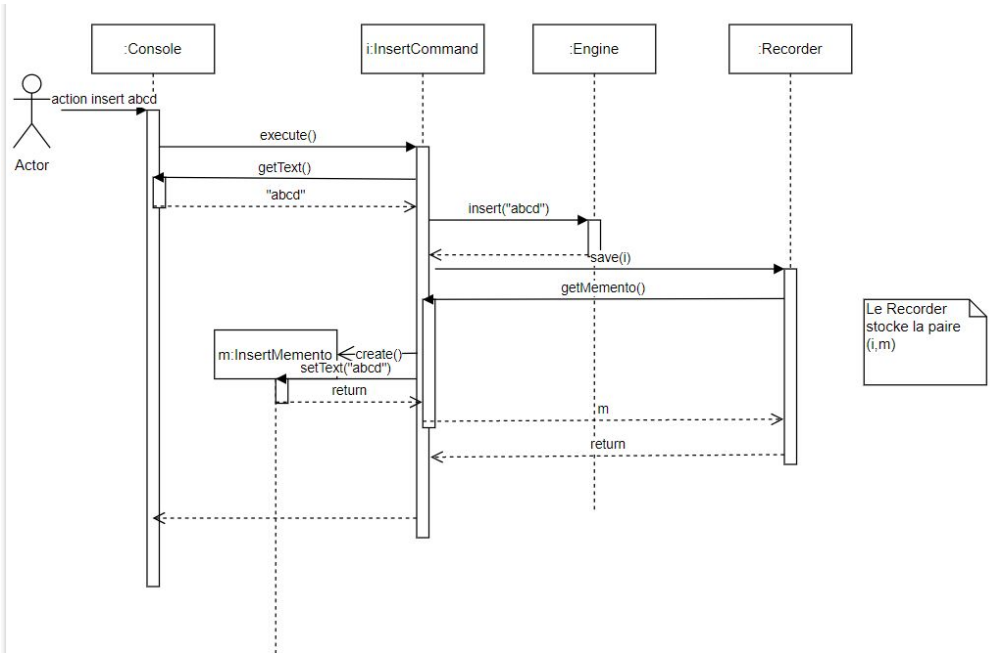
Dans cette version qui est une amélioration de la version 1, on devrait permettre à l'utilisateur d'enregistrer et de rejouer les actions enregistrées. Pour ce faire nous avons ajouté de nouvelles commandes (**Start**, **Stop** et **Replay**) et un nouveau receiver (**Recorder** et son implémentation **RecorderImpl**). Les deux premières commandes permettent de démarrer et d'arrêter l'enregistrement, et la commande Replay permet de rejouer les commandes enregistrées. Nous

avons aussi ajouté le patron de conception memento qui permet de stocker l'état des commandes qui seront enregistrées pour pouvoir les rejouer.

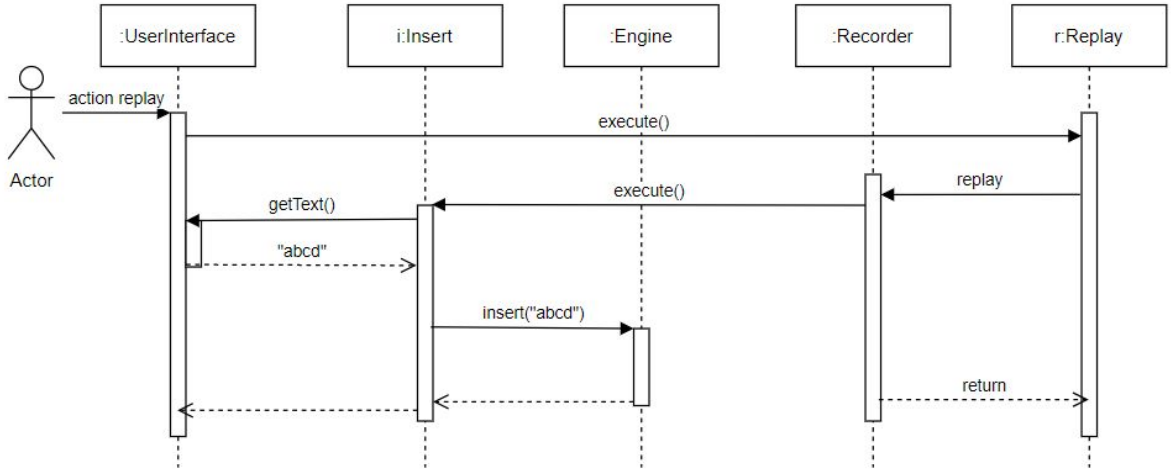




Diagrammes de classe des nouvelles commandes avec memento



Séquence de l'enregistrement de la commande Insert



Replay de la commande Insert

L'action de rejouer les commandes ne fonctionne pas dans certains scénarios. Prenons le scénario suivant par exemple :

L'utilisateur insère le contenu : "Salut", puis il démarre l'enregistrement, et fait un Select(0,5),

puis il fait la commande Delete. Quand l'action Select est rejouée une exception est levée parce que le contenu du buffer n'est plus le même qu'avant.

La figure ci-dessous montre un exemple d'exécution de la version 2, vous verrez bien que l'action Insert ("Comment vous allez?) est enregistrée puis rejouée.

Pour tester cette version, on procède de la même façon que la version 1, mais en se plaçant sur la branche version-2. Les nouvelles commandes à exécuter s'afficheront.

C. Version 3

Toujours dans la même logique, la version 3 consiste à rajouter à la version 2 les fonctionnalités défaire et refaire (le fameux undo et redo). Pour réaliser ces deux fonctionnalités, nous avons rajouté les commandes **RedoCommand** et **UndoCommand** qui permettent d'effectuer l'action correspondante, et un nouveau Caretaker (**UndoManager**) ce dernier permet de réaliser les actions de défaire et refaire. Étant contré par le temps, nous n'avons pu réaliser que le undo / redo des commandes Insert et Delete. Ces deux commandes jouent le rôle d'observateurs pour la classe UndoManager qui est le sujet, ce qui permettra de rétablir le texte supprimé si c'est le "undo" de Delete ou de supprimer le texte inséré si c'est le "undo" de Insert.

IV. Les tests unitaires

Pour s'assurer de la qualité du code fourni dans chacune des versions, nous avons réalisé un ensemble de cas de tests. Nos cas de tests se divisent en deux: l'Engine (Buffer, Selection) et les commandes.

1. Les tests de l'Engine

Pour chaque fonctionnalité du moteur, nous avons ajouté un ou plusieurs cas de tests.

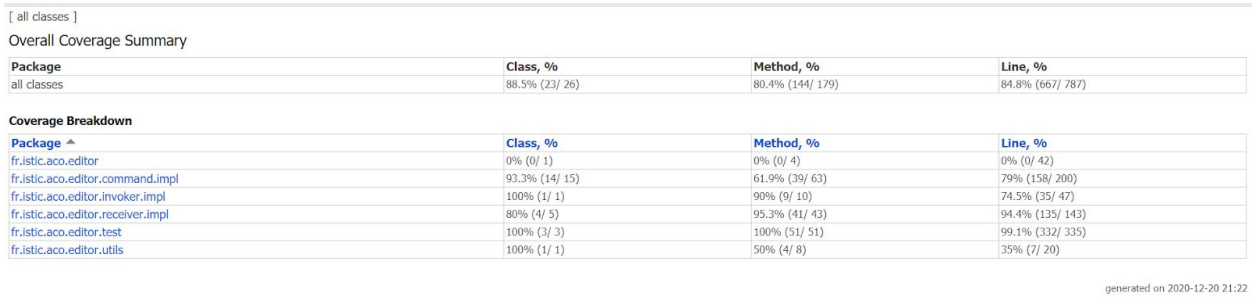
Étant donné que l'Engine joue le rôle de sujet (Subject) pour l'observateur **BufferChange**, nous avons aussi rajouté quelques cas pour tester la communication entre l'observateur (BufferChange) et le sujet (Engine).

2. Les tests des commandes

En plus des tests du moteur, nous avons ajouté un cas de test pour chaque commande concrète.

3. Synthèse des tests unitaires

L'objectif était d'avoir une couverture de 100 %. Mais vu que nous n'avons pas testé certaines classes que nous avons jugé moins critique comme EditorConfigutor ou QuitCommandpar exemple, on a eu une couverture d'environ 90% (statistiques visibles avec l'outil coverage de IntelliJ).



[all classes]

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88.5% (23/ 26)	80.4% (144/ 179)	84.8% (667/ 787)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
fr.istic.aco.editor	0% (0/ 1)	0% (0/ 4)	0% (0/ 42)
fr.istic.aco.editor.command.impl	93.3% (14/ 15)	61.9% (39/ 63)	79% (158/ 200)
fr.istic.aco.editor.invoker.impl	100% (1/ 1)	90% (9/ 10)	74.5% (35/ 47)
fr.istic.aco.editor.receiver.impl	80% (4/ 5)	95.3% (41/ 43)	94.4% (135/ 143)
fr.istic.aco.editor.test	100% (3/ 3)	100% (51/ 51)	99.1% (332/ 335)
fr.istic.aco.editor.utils	100% (1/ 1)	50% (4/ 8)	35% (7/ 20)

generated on 2020-12-20 21:22

Couverture des tests unitaires

Pour avoir plus de détails sur la couverture des tests, vous pouvez ouvrir le fichier index.html du dossier coverage qui se trouve à la racine du projet.

Conclusion

Notre travail s'achève par une proposition d'un mini éditeur de texte. Ce projet nous a permis de mettre en pratique les connaissances acquises durant les cours théoriques et pratiques de ACO.

Les différentes versions ainsi que le projet final sont disponibles sur gitlab.

Pour tester l'application, vous pouvez vous référer au **Readme** du projet disponible dans la branche master.

Lien vers la version 1

Lien vers la version 2

Lien vers la version 3

Lien vers la version finale