Sanaa Mairouch M1 MIAGE

Frédéric Rochard Groupe B

Compte rendu du TP Mini-éditeur

Encadrant TP : Noël Plouzeau

Table des matières

Présentation

Ce document présente le projet de conception et réalisation d'un mini éditeur de texte réalisé en TP d'ACO. L'objectif pédagogique de ce projet est de se familiariser à la conception d'application basé sur l'utilisation de patron de conception.

Objectif

L'objectif est de réaliser un mini-éditeur avec une interface java pour faciliter son utilisation, car celle-ci va réaliser plusieurs tâches qui seront : copier, couper, coller, enregistrer, refaire et défaire.

Afin de réaliser ce logiciel, nous avons fait trois versions :

- **Version 1 :** elle permet de réaliser les opérations de base d'un éditeur de texte (Copier, Coller, Couper, Sélectionner, Saisir)

- **Version 2 :** elle garde les fonctionnalités de la première version et on ajoute la possibilité d'enregistrer et de rejouer un enregistrement.

- **Version 3 :** elle garde les fonctionnalités de la deuxième version et on ajoute la possibilité de Défaire et Refaire une action.

V1

La version 1 de notre éditeur permet de réaliser plusieurs actions (couper, copier, coller, sélectionner, saisir). Il est composé d'une IHM et d'un moteur d'édition. Le moteur d'édition est composé de :

- Un buffer qui contient le texte de notre mini-éditeur
- Un presse papier qui contient le résultat des commandes *Couper* ou *Copier*.
- Une sélection caractérisée par sa position de début et sa longueur

Design Pattern command

L'IHM déclenche une liste de commandes à effectuer dans le moteur d'édition. Le design pattern *Command* est de type comportemental et encapsule la notion d'invocation. Il permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même. Ce patron de conception est souvent utilisé dans les interfaces graphiques où, par exemple, un élément de menu peut être connecté à différentes Commandes de façon que l'objet d'élément de menu n'ait pas besoin de connaître les détails de l'action effectuée par la Commande. Nous avons donc décidé d'implémanter le design pattern Command avec les rôles ci-dessous :

- Receiver : MoteurEdition
    - Cette interface réalise l'action sur demande de la commande concrète avec une opération associée.
- Invoker : Ihm
    - On appelle les commandes concrètes depuis l'IHM suivant l'action exécutée par l'utilisateur. Notre Ihm détecte les évènements avec différents listener et exécute la commande concrète associée à l'évènement.
- Client : Editeur
    - L'éditeur configure l'invoker (Ihm) en créant et configurant les commandes concrètes.
- Command
    - Interface commune aux commandes concrètes
- Commandes concrètes : (Copier, Coller, Couper, Saisir, Sélectionner)
    - Il y a une classe par commande concrète. Elles Provoquent l'exécution d'une opération par le moteur d'édition (receiver).

Design Pattern observer

Certaines commandes comme *Couper* ou *Coller* modifient l'état du moteur d'édition. L'IHM doit donc être informé du changement d'état du moteur d'édition.

Le patron de conception Observer est utilisé en programmation pour envoyer un signal à des modules qui jouent le rôle d'observateur. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les « observables »).

Nous avons donc décidé d'implémenter le design pattern *Observer* pour répondre à la problématique de mise à jour de l'IHM.

- Concrete Observer : Ihm
    - L'Ihm observe le moteur d'édition. Elle implémente l'interface *Observer*.
- Subject : MoteurEditionImpl
    - Le moteur d'édition est observé par l'IHM. Elle hérite de la classe Observable. On ajoute l'Ihm comme observer du moteur d'édition lors de l'instanciation d'un nouvel éditeur. A chaque exécution des méthodes Couper ou Coller, une notification est envoyée au moteur d'édition (observer) pour l'informer du changement d'état.

Diagramme de classe

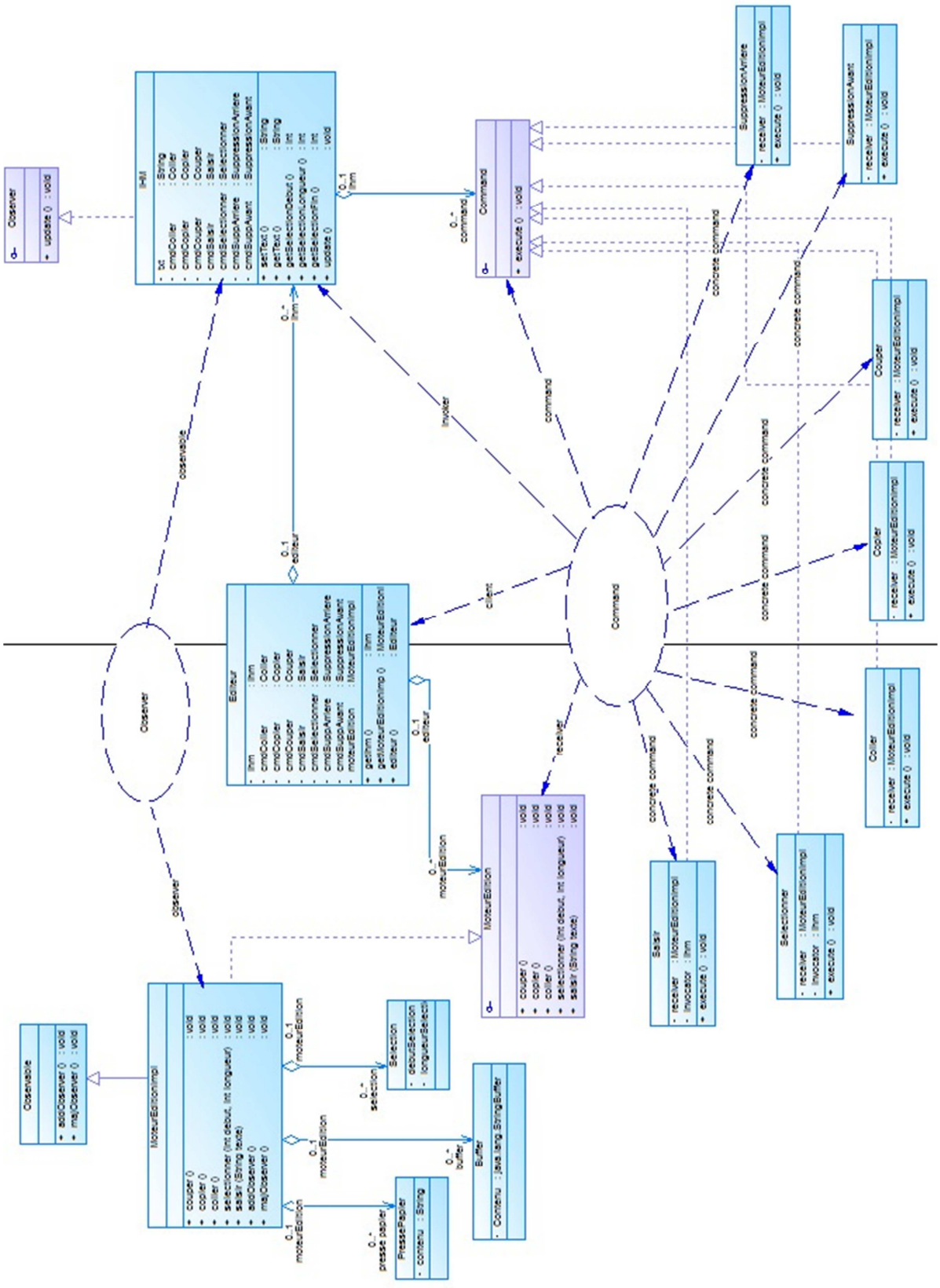
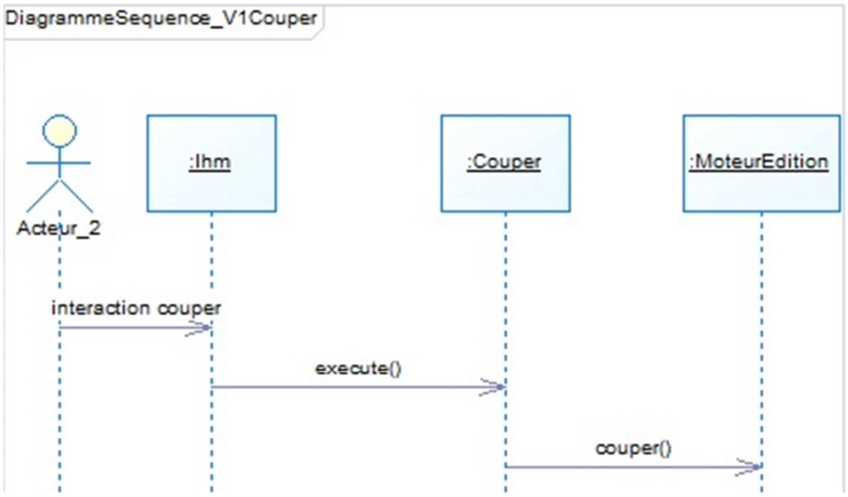
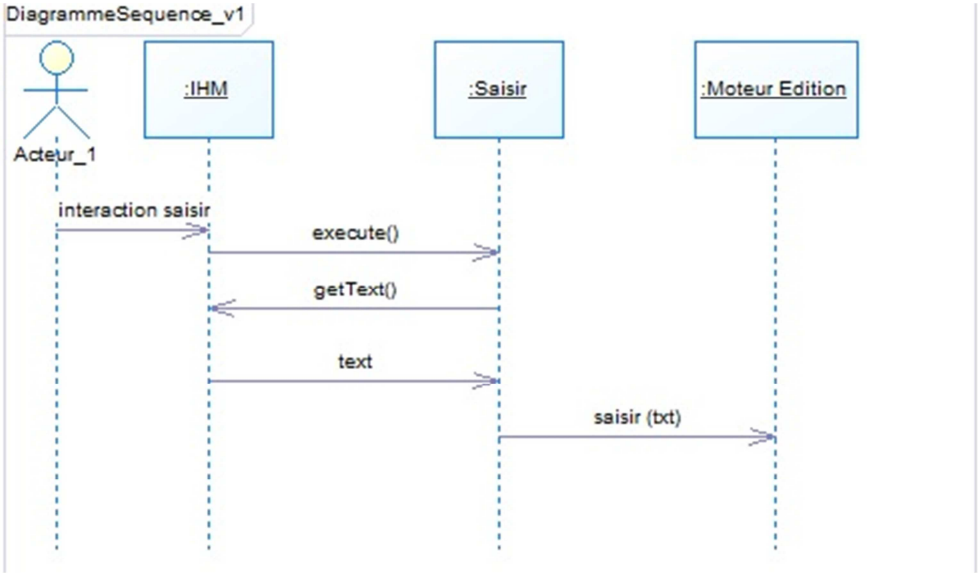


Diagramme de séquence

Commande simple : Couper



Commande avec paramètre : Saisir



Cas de test

Nous avons réalisé 32 tests unitaires JUnit pour cette version sur le moteur d'édition, le Buffer, le presse papier et l'éditeur.

Les tests des commandes sont réalisés avec des assertions sur le buffer, le presse papier ou la sélection notamment.

V2

Cette version permet d'enregistrer une suite de commandes et de la rejouer. Il faut donc mémoriser les commandes dans une liste ordonnée ainsi que certains paramètres comme le texte à saisir ou les valeurs de la sélection.

Le design pattern mémento permet d'enregistrer et de restaurer l'état d'un objet en respectant le principe d'encapsulation. Nous décidons donc d'implémenter ce patron de conception dans notre version 2.

Les commandes de la V1 doivent donc conserver le même comportement et être mémorisé. Nous décidons donc de créer des commandes enregistrables héritant des commandes de la V1 ce qui nous permet de toucher aux commandes de la V1. Nous ajoutons à ces commandes des fonctions pour les mémoriser et les rejouer.

L'enregistrement peut être actif ou non au moyen de commande *Arrêter* et *Demarrer* enregistrement. Nous utiliserons le design pattern *Command* déjà utilisé en V1 pour implémenter ces commandes.

Nous avons rencontré des difficultés lors de l'implémentation de cette version pour maîtriser le comportement de l'interface graphique. En effet, l'objet Swing `JTextArea` a sa propre gestion de curseur, ce qui a conduit à des comportements inattendus. Par exemple, la perte de focus entraîne une réinitialisation au début de la zone de texte. L'objectif pédagogique de ce projet étant d'appréhender les patrons de conception, **nous avons donc pris le parti de démontrer le fonctionnement de cette version au travers de nos cas de test.**

Design Pattern command

L'IHM déclenche les commandes *Demarrer, Arreter* et *Rejouer* à effectuer dans l'enregistreur de commande. Nous avons implémenté le design pattern *Command* avec les rôles ci-dessous :

- Receiver : *EnregistreurV2*
  - Cette interface réalise l'action sur demande de la commande concrète avec une opération associée.
- Invoker : *Ihm*
  - On appelle les commandes concrètes depuis l'IHM suivant l'action exécutée par l'utilisateur. Notre Ihm détecte les évènements avec différents listener et exécute la commande concrète associée à l'évènement.
- Client : Editeur
  - L'éditeur configure l'invoker (Ihm) en créant et configurant les commandes concrètes.
- Command
  - Interface commune aux commandes concrètes
- Commandes concrètes : (*Demarrer, Arreter* et *Rejouer*)

- Il y a une classe par commande concrète. Elles Provoquent l'exécution d'une opération par l'enregistreur (receiver).

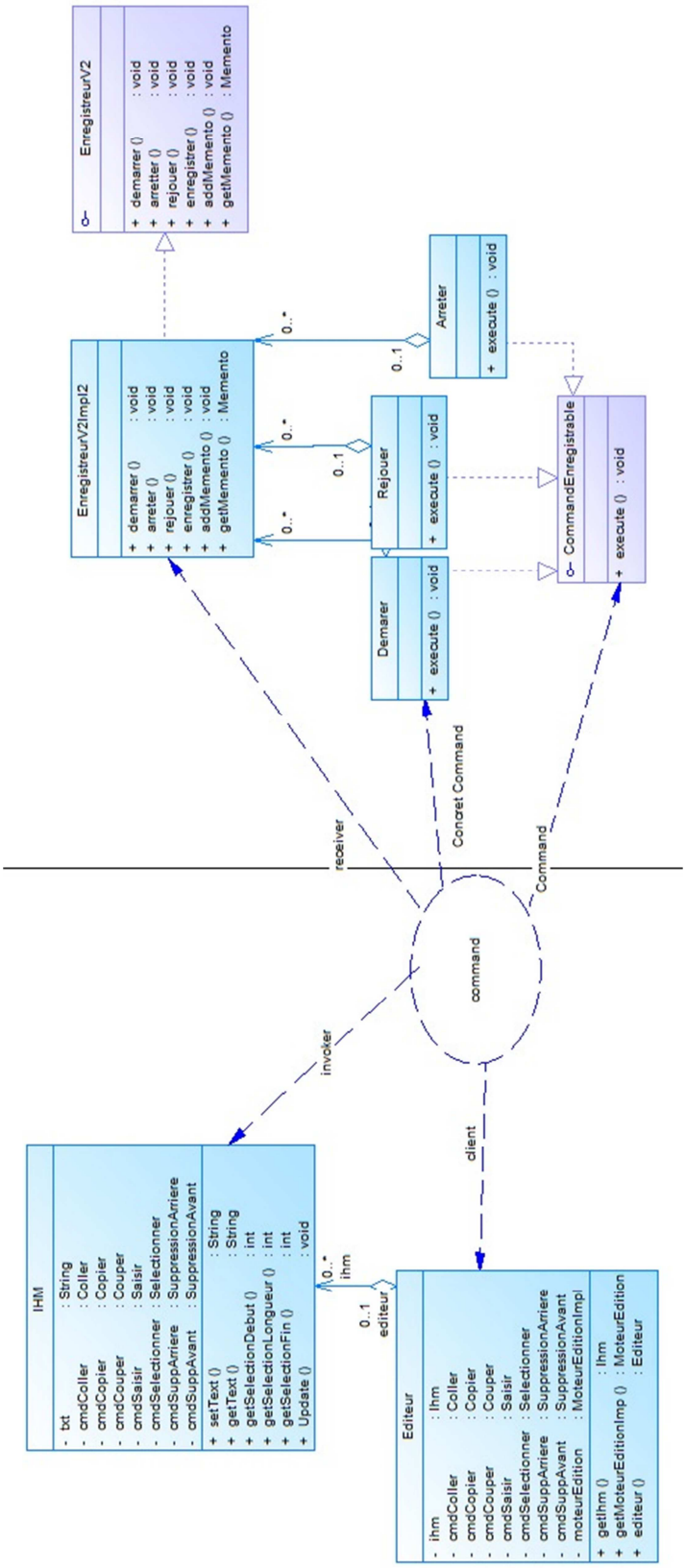
Design Pattern memento

La commande *Rejouer* doit avoir accès à une liste ordonnée de commandes enregistrés. Nous utiliserons donc le design pattern Memento pour réaliser cette implémentation avec les rôles ci-dessous :

- Caretaker : EnregistreurV2
  - L'enregistreur de la V2 contient la liste ordonnée des commandes enregistrées sous forme de memento.
- Originator : Interface *commandEnregistrableV2*
  - Cette interface est implémentée par les commandes de la V2.
- Memento : Interface Memento
  - C'est le type utilisé pour les commandes enregistrées par l'enregistreur (Caretaker). Cette interface est vide car le caretaker ne doit rien connaître des objets qu'il enregistre.
- Memento concrets : MementoCopier, MementoCouper, MementoColler, MementoSaisir, MementoSelectionner
  - Ces classes implémentent l'interface Memento. Chaque memento concret correspond à une commande enregistrable de la V2.

Diagramme de classe

Design pattern Command pour la v2



Design pattern Memento pour la v2

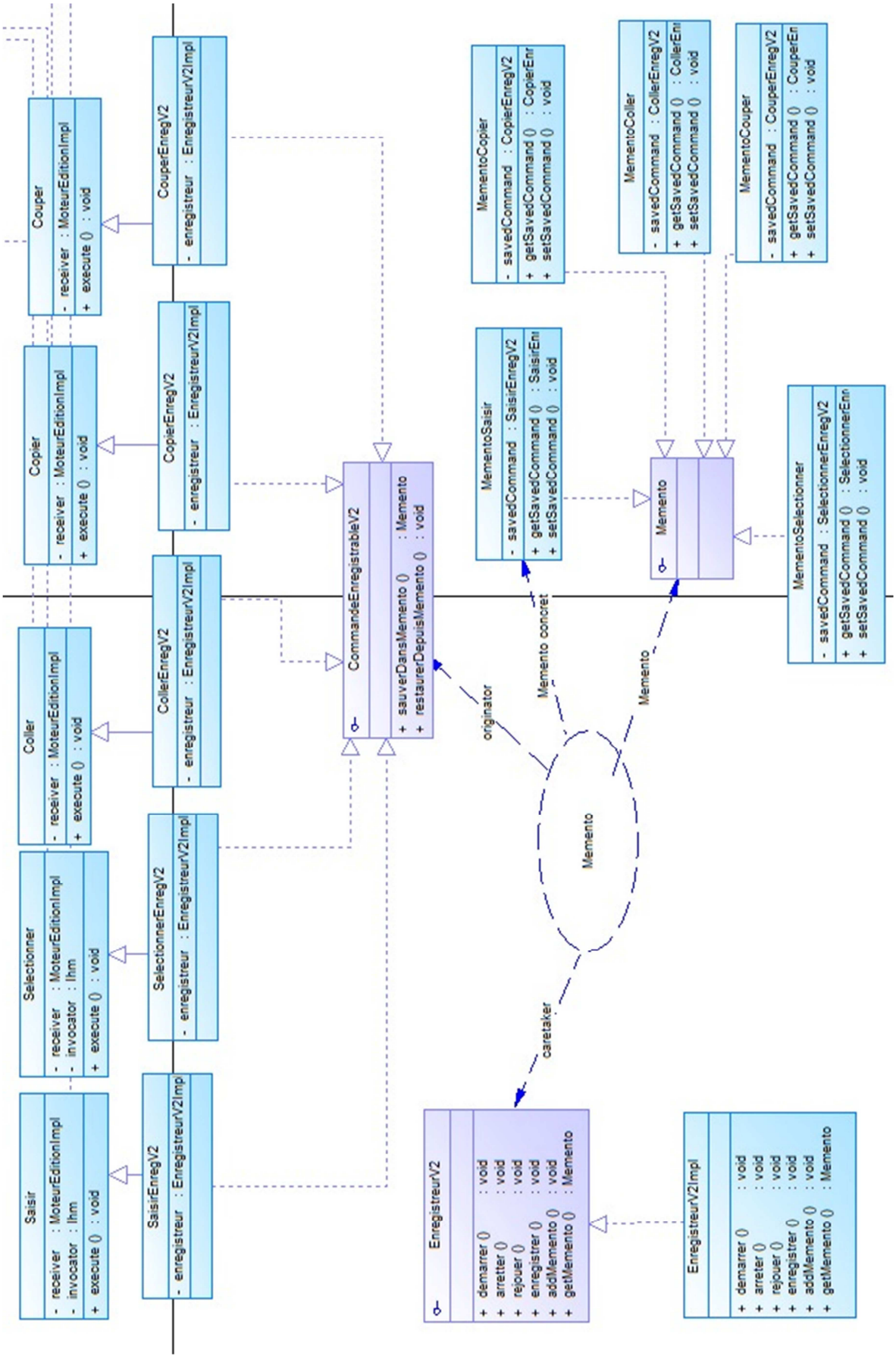


Diagramme de séquence

Diagramme de séquence pour démarrer et couper

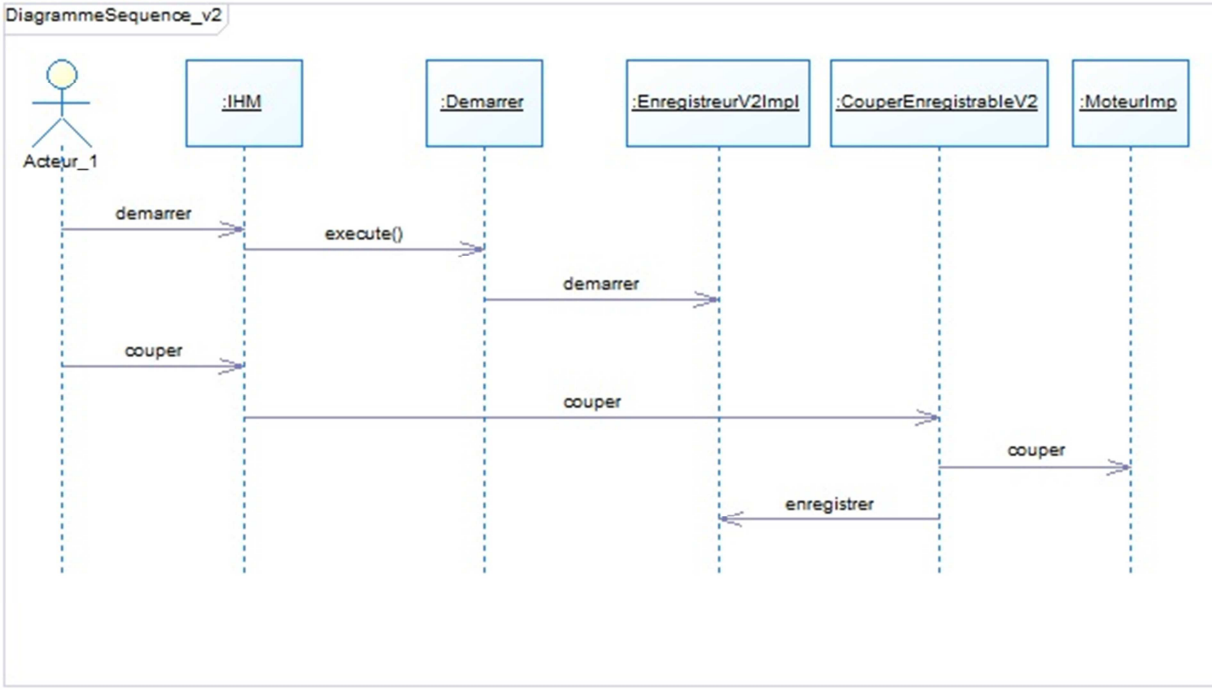
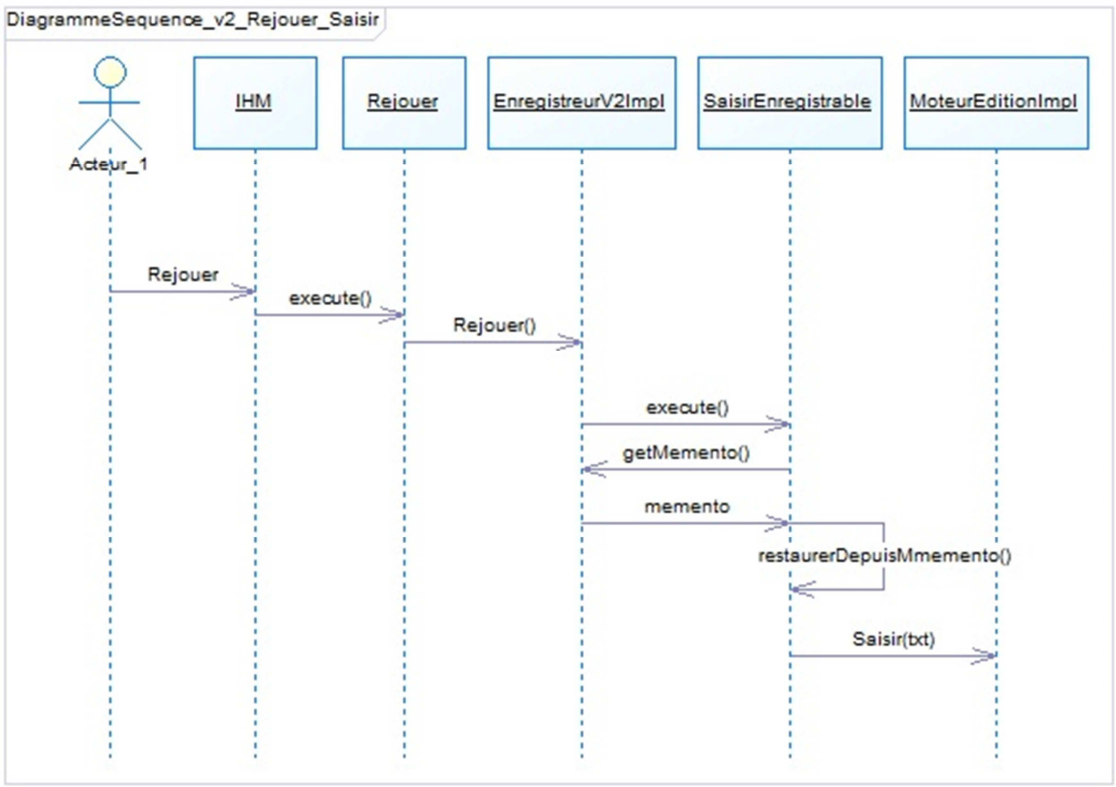


Diagramme de séquence pour *Rejouer*



Cas de test

Nous avons réalisé des tests sur les nouvelles commandes de cette version (*Demarrer*, *Arreter* et *Rejouer*). Nous avons également réalisé plusieurs scénarii de test dont celui vu en cours.

V3

La version 3 du mini-éditeur doit permettre d'ajouter de nouvelles commandes :

- La commande *Défaire* doit permettre de rétablir le moteur d'édition à l'état antérieur à la dernière commande.
- La commande *Refaire* doit permettre de rétablir le moteur d'édition à l'état antérieur à la commande défaire.

Si une autre action que défaire est effectuée, il n'y a plus de commande *refaire* disponible.

Cette version doit donc permettre à l'IHM d'ordonner l'exécution d'une commande à l'enregistreur des états du moteur d'édition. On utilisera encore une fois le design pattern Command.

Pour l'enregistrement des états du moteur à défaire ou à refaire, nous utiliserons le design pattern *Memento*. Nous avons choisi d'nregistrer à chaque modification du moteur d'édition la chaîne contenue dans le buffer et la sélection.

Design Pattern command

L'IHM comporte deux nouvelles commandes *défaire* et *refaire* qui doivent être exécutées par l'enregistreur d'état du moteur d'édition. Nous utiliserons le design pattern *Command* pour les implémenter avec les rôles ci-dessous :

- Receiver : *EnregistreurV*3
    - Cette interface réalise l'action sur demande de la commande concrète avec une opération associée.
- Invoker : *Ihm*
    - On appelle les commandes concrètes depuis l'IHM suivant l'action exécutée par l'utilisateur. Notre Ihm détecte les évènements avec différents listener et exécute la commande concrète associée à l'évènement.
- Client : Editeur
    - L'éditeur configure l'invoker (Ihm) en créant et configurant les commandes concrètes.

- Command
    - Interface commune aux commandes concrètes
- Commandes concrètes : (*Défaire* et *Refaire*)
    - Il y a une classe par commande concrète. Elles Provoquent l'exécution d'une opération par l'enregistreur V3(receiver).

Design Pattern memento

Les commandes *défaire* et *refaire* doivent avoir accès à une liste ordonnée d'état enregistré du moteur d'édition. Nous utiliserons donc le design pattern Memento pour réaliser cette implémentation avec les rôles ci-dessous :

- Caretaker : EnregistreurV3
    - L'enregistreur de la V3 contient la liste ordonnée des états du buffer sous forme de memento.
    - Il possède deux piles de memento :
        - une pile pour les commandes à défaire
        - une pile pour les commandes à refaire
    - Lorsque la commande défaire est appelée, le memento correspondant est retiré de la pile défaire et est ajouté en haut de la pile refaire. Si une autre commande est appelée, la pile Refaire est vidée.
- Originator : Interface *commandEnregistrableV3*
    - Cette interface est implémentée par les commandes de la V3 :
        - CouperEnregistrableV3
        - CollerEnregistrableV3
        - SaisirEnregistrableV3
        - La commande copier n'agit pas sur le buffer, elle n'implémente donc pas l'interface CommandEnregistrableV3.
- Memento : Interface Memento
    - C'est le type utilisé pour les commandes enregistrées par l'enregistreur (Caretaker). Cette interface est vide car le caretaker ne doit rien connaître des objets qu'il enregistre.
- Memento concrets : MementoMoteurEdition
    - Cette classe implémente l'interface Memento. Ce memento concret contient le texte du buffer, ainsi que la position de début et la longueur de la sélection.

Diagramme de classe

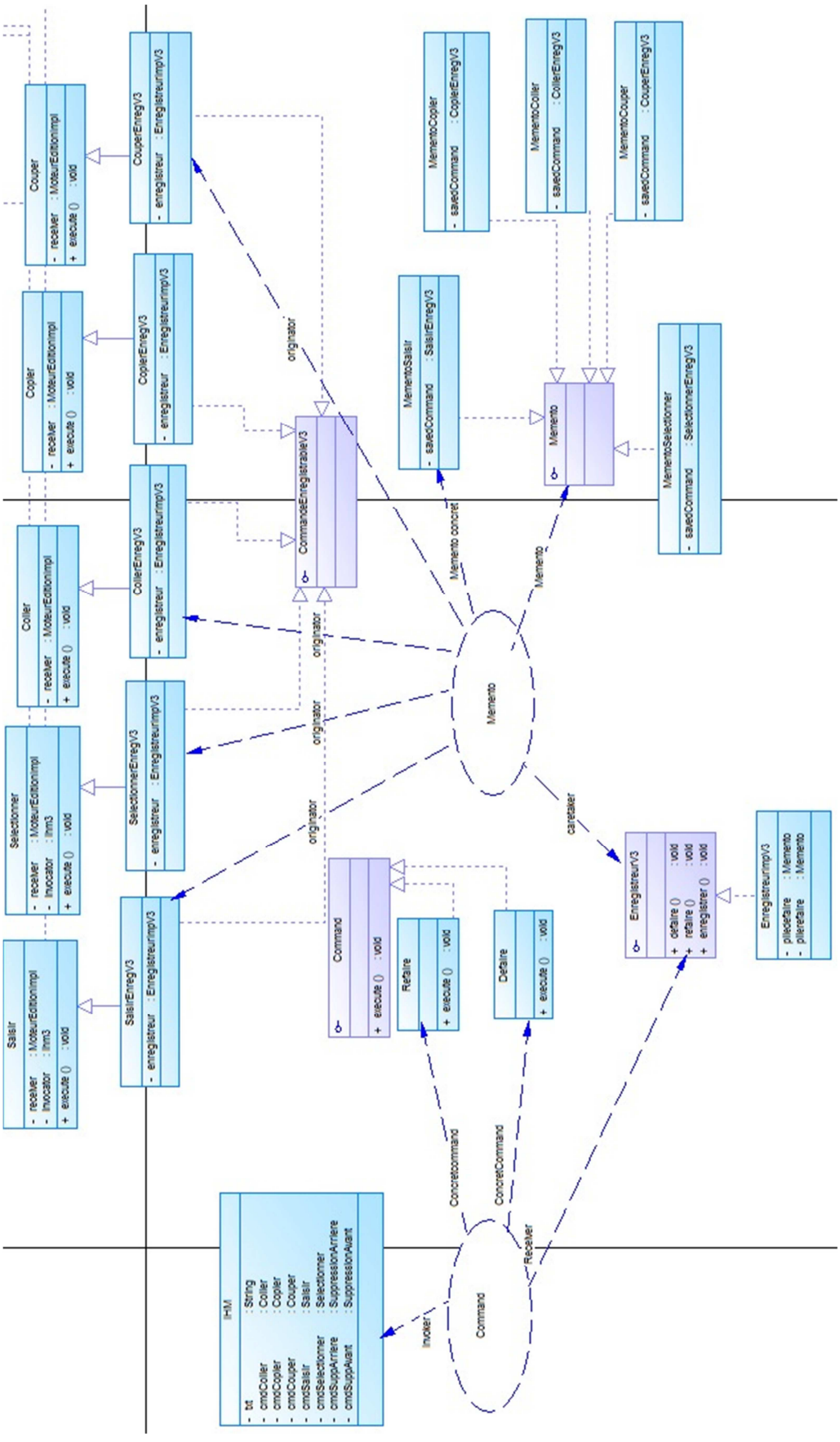


Diagramme de séquence

Diagramme de séquence pour *Couper*

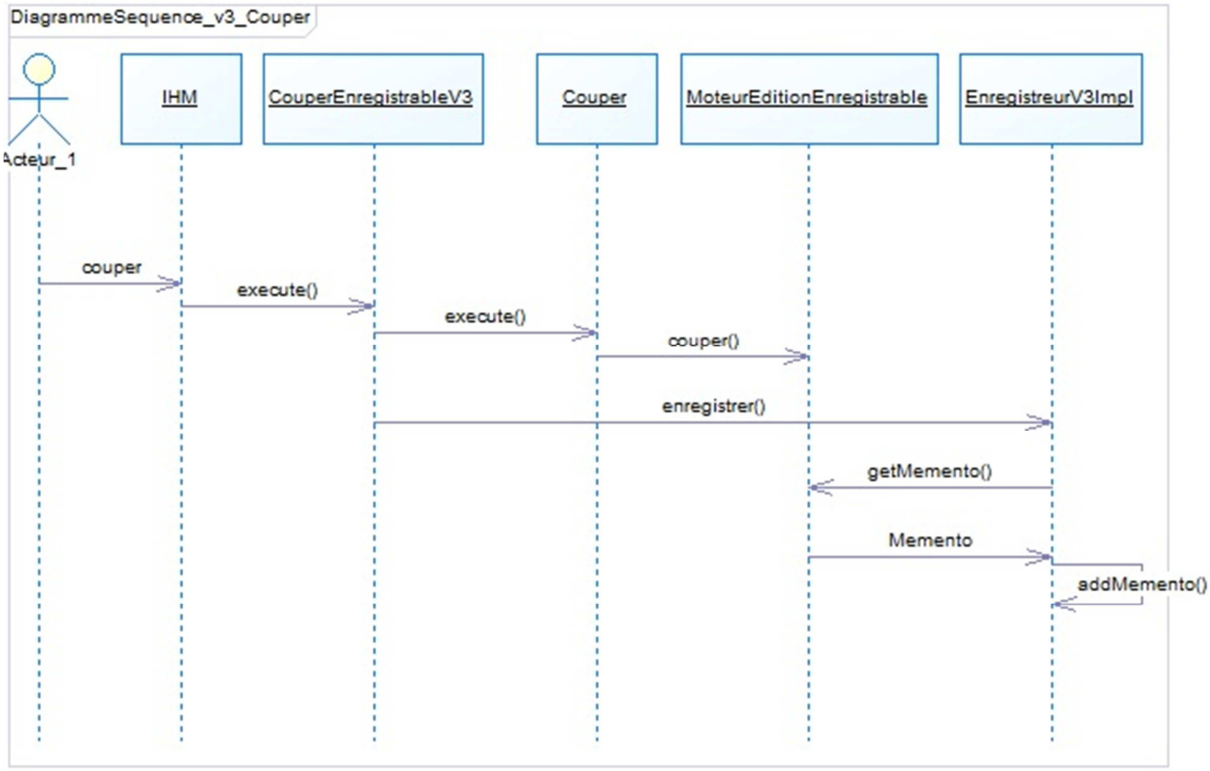


Diagramme de séquence pour *Defaire*

Cas de test

Nous avons réalisé plusieurs scenarii de test afin de démontrer le fonctionnement des nouvelles fonctionnalités de la V3.

Conclusion

La réalisation d'un mini-éditeur de texte nous a permis de nous familiariser avec l'implémentation de patrons de conception. L'ajout de commande au fil des différentes versions a ainsi pu être facilité en utilisant le même mode opératoire. La structuration de notre programme avec les patrons de conception a également permis une meilleure réutilisabilité de notre code au fil des versions.

Nous avons également utilisé le logiciel PowerAMC pour la réalisation des diagrammes UML de classe et de séquence.

Nous avons également documenté notre projet avec l'outil *Javadoc*. Le format html généré avec cet outil permet une lecture claire de notre application. Cela s'est avéré très utile dans le cadre de ce travail collaboratif ou lorsqu'il a fallu revenir sur notre code.

Ce projet a également été l'occasion de produire un ensemble de tests unitaires à l'aide du Framework JUnit afin de démontrer la robustesse de notre code. Ces tests unitaires ont également permis d'assurer la non-régression de notre programme au cours du développement de nouvelles versions.