Rapport de projet

ACO - M1 Info GL2

Mullier Antoine
Sadok Romain

Encadrant de TP : *Noël Plouzeau*

Table des matières

Introduction

Ce rapport présente la réflexion, la conception et la réalisation du projet “Mini Éditeur” dans le cadre de l’enseignement ACO. Pour expliquer au mieux notre raisonnement, nous suivrons l’ordre des versions demandées en abordant les concepts UML utilisés et les choix que nous avons fait pour arriver au résultat attendu.

Version 1 : L’éditeur de base

1.1 - Travail demandé

Pour la version 1, l’éditeur qui nous a été demandé devait remplir les fonctions de base d’un éditeur de texte c’est à dire :

- L’insertion de texte
- La sélection
- La commande Copier
- La commande Couper
- La commande Coller
- La suppression de la sélection

En analysant le principe de base du mini-éditeur nous avons pu déterminer en TD qu’il fallait **un moteur d’édition** du mini-éditeur qui comporte les opérations de la **mécanique interne**. Nous avons donc choisi d’implémenter le moteur de la façon suivante :

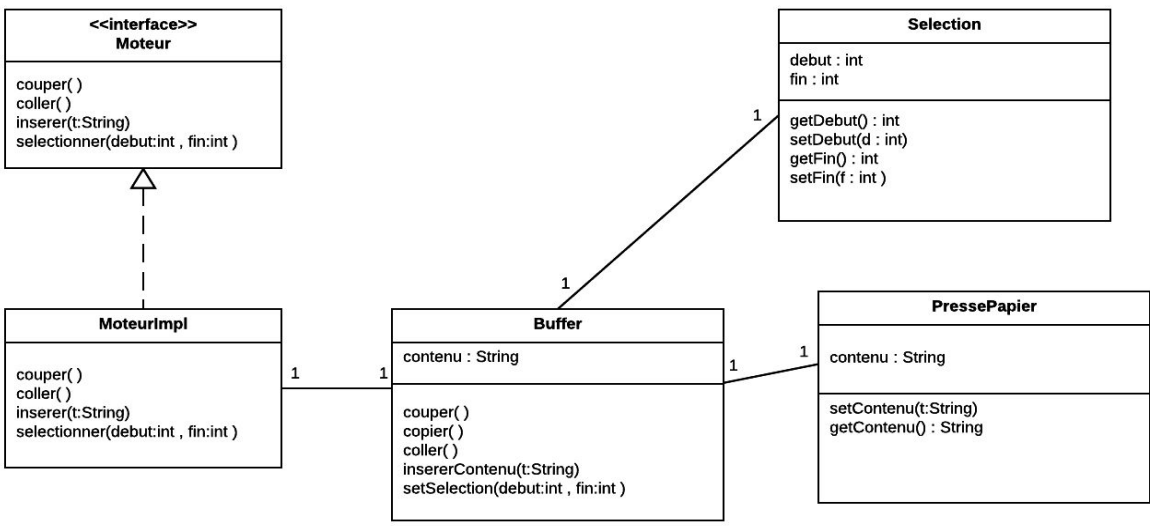

Figure 1 : Moteur d’action du mini-éditeur

1.2 - Analyse

Le patron de conception “**Commande**” vu en cours nous permet de concevoir cette version du mini-éditeur. On appliquant le patron de conception Commande, nous obtenons le diagramme des classes suivant :

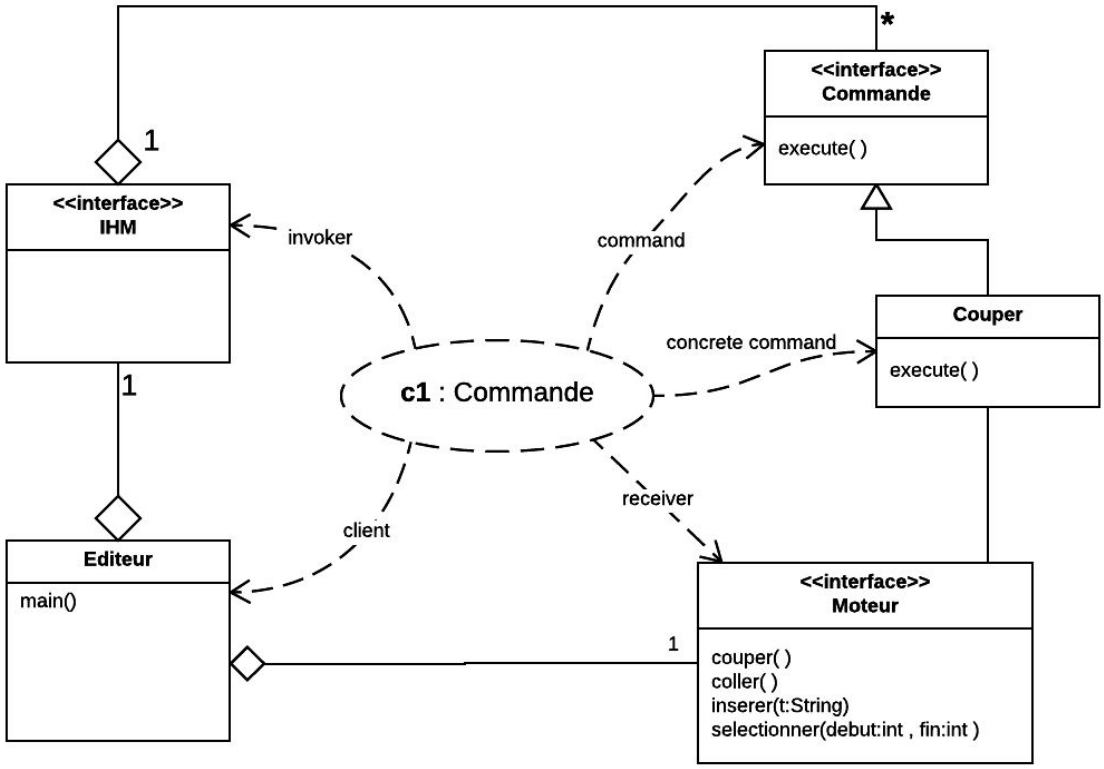

Figure 2 : Patron de conception Commande pour la version 1

Avec ce choix de conception nous allons pouvoir ajouter des commandes en ne modifiant que peu de code on peut ainsi ajouter des fonctionnalités sans devoir “casser” ce qui à déjà été implémenté.

1.3 - Scénarios

Ainsi en analysant le scénario d’un appel de commande Couper on obtient le diagramme de séquence suivant :

Figure 3 : Diagramme de séquence de l’action Couper de l’utilisateur

Dans le cas précédent la fonction de la commande ne prend pas de paramètre mais dans le cas d'insérer texte il faut que l'objet **Inserer** récupère depuis l'IHM le texte saisi. On obtient donc la variante de diagramme de séquence suivant:

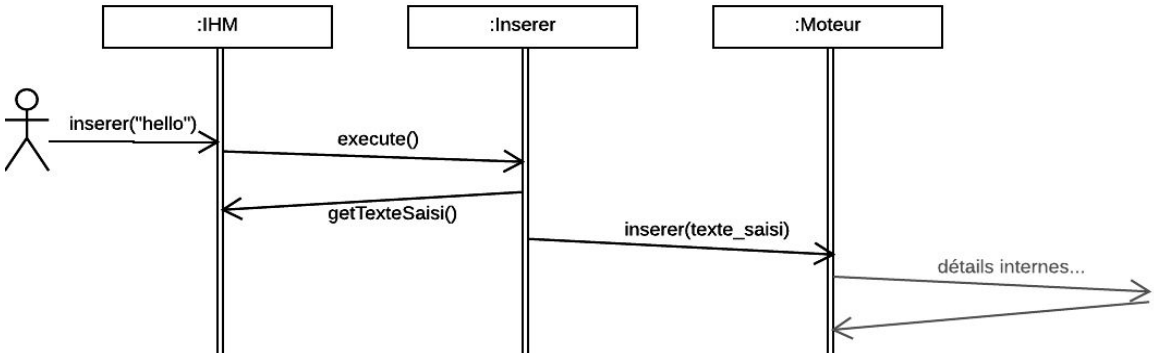

Figure 4 : Diagramme de séquence de l'action Insérer de l'utilisateur

1.4 - Tests

Les tests effectués par la version 1 traitaient principalement de la mécanique interne du moteur pour voir si les différentes commandes implémentées par la classe *MoteurImpl* faisaient des traitements correct sur le moteur.

De plus, les tests de la version nous ont permis d'effectuer les premiers scénarios de test avec l'*IHM*. Ces tests nous ont permis de valider les fonctionnalités de la version 1.

Version 2 : Enregistrement

2.1 - Travail demandé

Pour la version 2 le travail demandé est d'ajouter trois commandes : **Démarrer enregistrement**, **Arrêter enregistrement** et **Rejouer enregistrement**.

La conception de cette version nous a amené à faire certains choix arbitraires sur le fonctionnement de notre mini-éditeur:

- **La commande de sélection n'est pas enregistrable**

Lors de l'enregistrement le buffer peut faire 150 caractères, rejouer l'événement "*Selection(142,145)*" sur un buffer de 3 cases n'aurait pas de sens.

- **Conséquence : Les commandes de presse papier se rejouent avec la sélection actuelle**

Lors de l'enregistrement on a fait un copier/coller de la sélection (0,1)
si, après la fin de l'enregistrement, on fait une autre sélection et que l'on rejoue l'enregistrement le copier/coller se fera avec la sélection courante.

2.2 - Analyse

De prime abord nous aurions pu sauvegarder une liste de commandes qui l'on aurait put ré-exécuter lors l'appel à la commande rejouer. Cependant, le cas de la commande **Inserer** pose problème car il faut sauvegarder un état dans lequel la commande à été exécutée (le texte qui a été inséré.

Comme nous l'avons vu en TD nous allons utilisé **le patron de conception Mémento** qui nous permet de gérer la sauvegarde d'un état.

De plus, il nous faut ajouter des commandes concrètes au patron de conception **c1:Commande**, pour toutes les commandes d'actions de l'enregistreur (*définies en 2.1*).

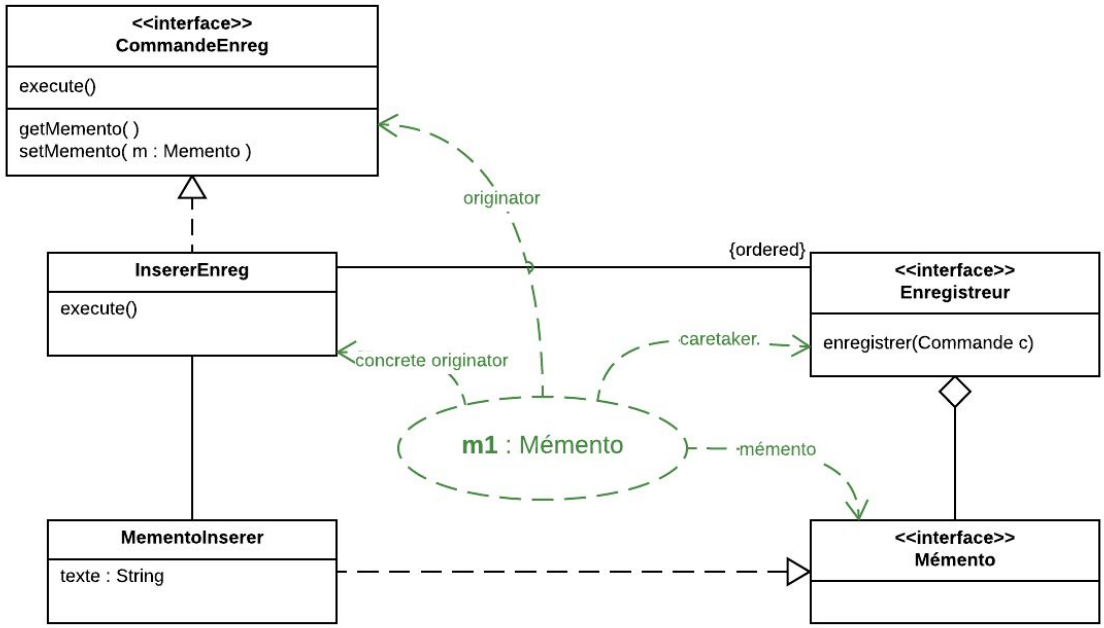

Figure 3 : Patron de conception Mémento appliqué pour la gestion de l'enregistrement

Pour que l'IHM sache distinguer l'utilisation de la commande *Inserer* de la commande *InsererEnreg* on introduit un attribut **en_cours_enregistrement** - de type booléen - dans la classe IHM.

> ***Remarque :***
> Un axe d'amélioration de ce projet pour être la mise en place d'un patron de conception *Observer* sur l'IHM pour qu'elle soit notifiée de tout changement de l'état de l'enregistreur.

2.3 - Scénarios

Pour bien comprendre le rôle du mémento dans cette version nous allons établir deux scénarios :

- Enregistrement de la commande Insérer
- Rejouer l'enregistrement de la commande Insérer

Remarque : Les autres commandes enregistrables sont des cas plus triviaux que la commande d'insertion de texte. En montrant comment se passe la commande Insérer on couvre le cas le plus complexe.

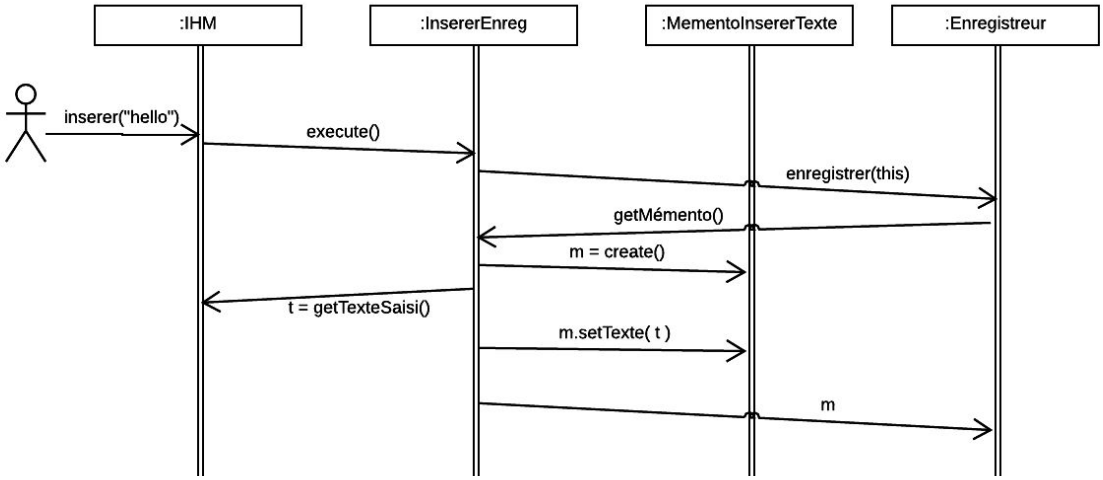

Figure 5 : Diagramme de séquence de l'enregistrement de la commande Insérer

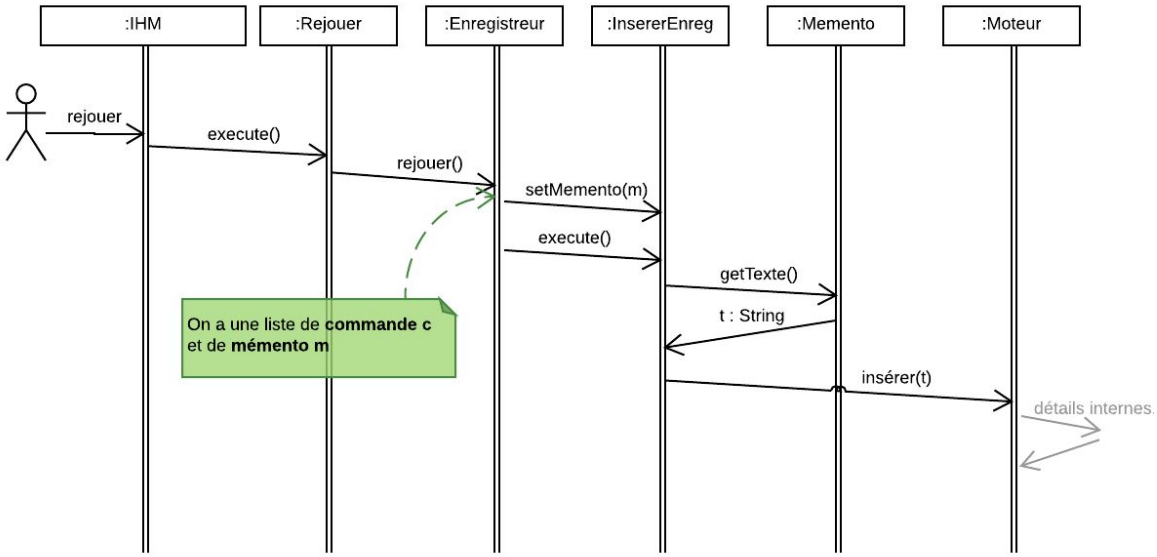

Figure 6 : Diagramme de séquence de l'action Rejouer d'une commande Insérer

2.4 - Tests

Les tests de la version 2 nous ont permis d'établir différents scénarios que nous avons rejoué après avoir réinitialisé le moteur. Le but de cette méthode était de pouvoir comparer les deux moteurs pour voir si lorsque l'on rejoue les actions effectuées donne le même résultat.

Version 3 : Défaire/Refaire

3.1 - Travail demandé

Pour la version 3 le travail demandé est d'ajouter deux commandes **Défaire** et **Refaire**. Le principe est de gérer l'historique du mini-éditeur en enregistrant donc les différents états du buffer.

3.2 - Analyse

3.2.1 - États du buffer

Pour arriver au résultat final souhaité avec la sauvegarde des états du buffer nous allons utiliser schématiquement deux piles : *pile_defaire*, *pile_refaire*.

Pile défaire : pile composée des états du buffer passés
Pile refaire : pile composée des états du buffer *"du futur"*[1]

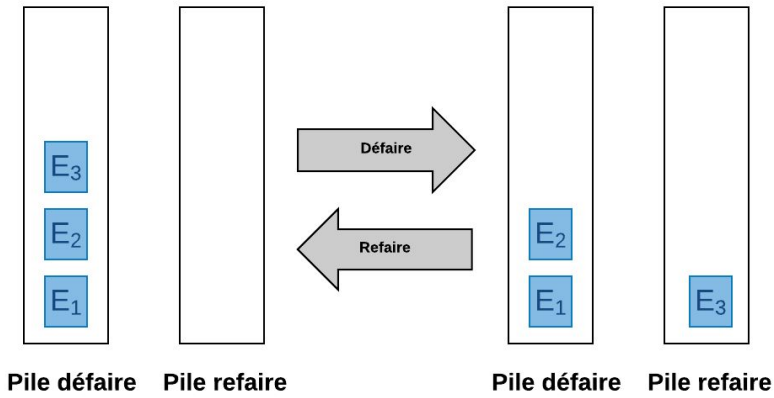

Figure 7 : Actions de défaire et refaire sur les piles de sauvegardes de l'état du buffer

Nous estimons que le contenu du presse papier ne sera pas impacté par l'action de défaire refaire. En effet, nous voulons que l'utilisateur puisse revenir en arrière et coller un texte qui était déjà présent dans le presse papier. Un état de buffer devra contenir le contenu et la sélection de l'état.

3.2.2 - Optimisation du buffer

Sachant qu'un état du buffer contient le contenu du buffer et la sélection du buffer il problème survient : **l'optimisation**. En effet, dans le cas où notre mini-éditeur soit utilisé pour traiter de gros fichiers, par exemple 1Go, **un état du buffer ferait à lui seul 1Go**. En faisant 50 opérations on arriverait à **une demande de mémoire de 50 Go !!!**

[1] Événements passés qui ont été défaits sans suite de modifications

Allouer 50 Go de mémoire étant impensable il nous faut trouver une alternative à la sauvegarde de tous les états du buffer tel que décrit précédemment.

Nous avons trouvé deux alternatives possibles au problème d'optimisation.

Sauvegarde d'un état toutes les n opérations :

Il s'agit de sauvegarder toutes les commandes effectuées et de quelques états pour pouvoir revenir à l'état demandé par la commande Défaire ou Refaire.

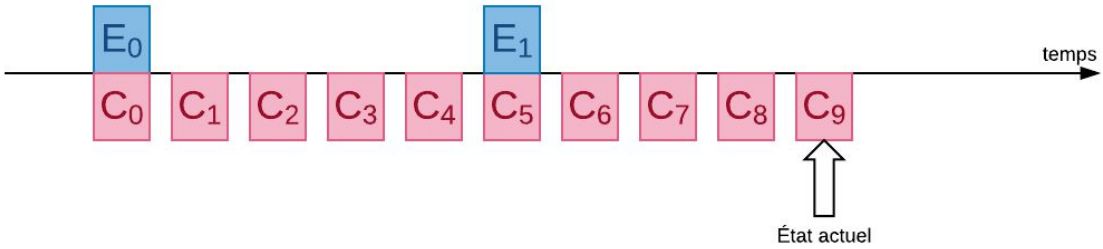

Figure 8 : Illustration de la sauvegarde partielle des états

Ainsi pour Défaire il faut d'abord revenir au dernier état enregistré avant d'exécuter les commandes jusqu'à l'état demandé par l'utilisateur.

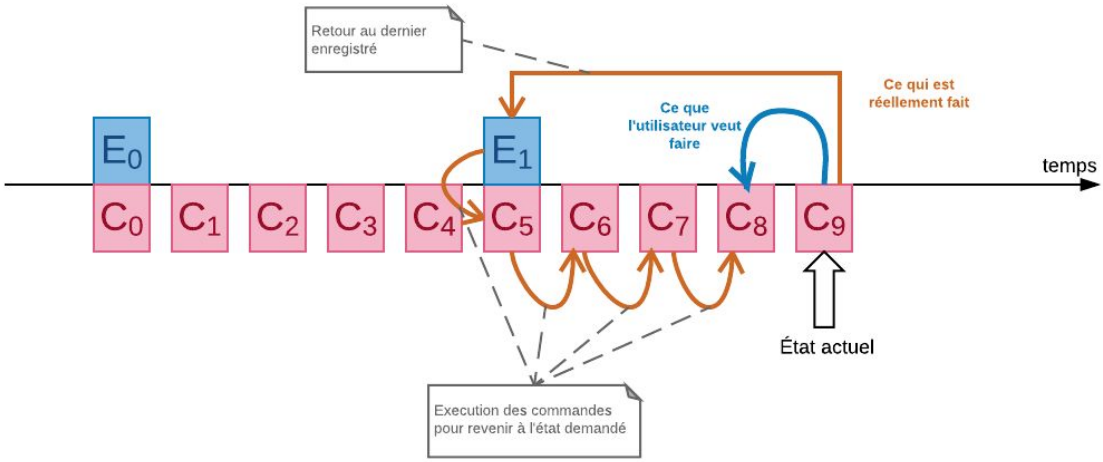

Figure 9 : Illustration de l'exécution d'une action Défaire avec une sauvegarde partielle des états

Approche plus fine : Suite de fibonacci :

On sait que l'utilisateur fait le plus souvent peu de défaire/refaire. Par exemple, si l'utilisateur à fait 100 opérations il est *peu probable*[2] qu'il souhaite retourner à l'état de sa première opération.

En reprenant le principe de la sauvegarde partielle des états détaillée dans le paragraphe précédent et en suivant la suite de fibonacci pour choisir les états à sauvegarder on a besoin d'encore moins d'états pour couvrir tous les états possibles.

[2] Dans le sens où l'utilisateur cherche rarement à défaire 100 actions.

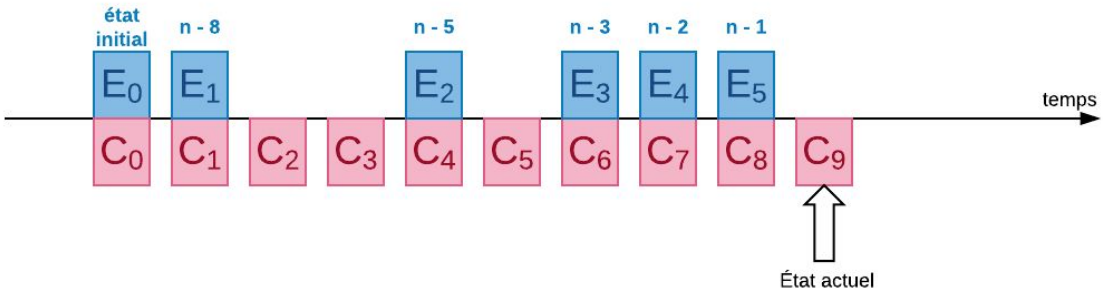

Figure 10 : Illustration de l'approche avec la suite de Fibonacci

Cependant cette méthode nécessite une mise à jour à chaque commande exécutée pour mettre à jour les états n-1, n-2, n-3 … ce qui peut poser un problème d'optimisation en temps d'exécution.

3.2.3 - Méthode choisie pour la gestion des états

Nous avons choisi la méthode de sauvegarde partielle des états en prenant **n=5**. Pour implémenter cette sauvegarde nous allons mettre en place un patron de conception Memento. La gestion de la méthode de sauvegarde des états sera faite par le **caretaker GestionDefaireRefaire**.

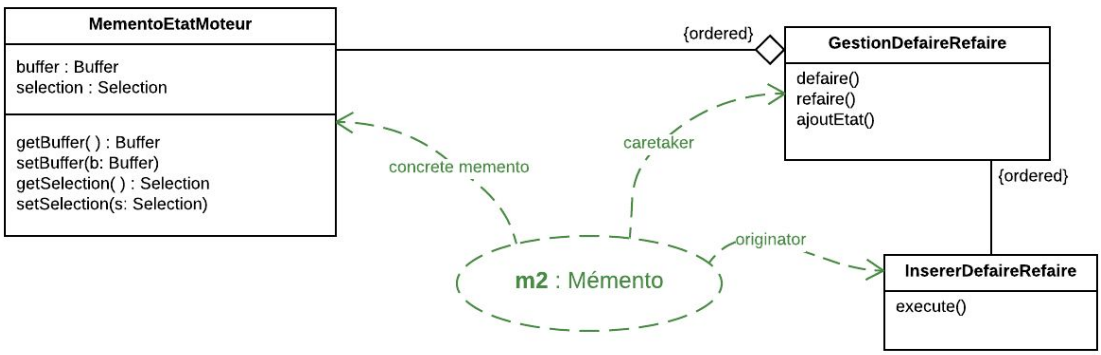

Figure 11 : Patron de conception Memento appliqué à la version 3

Nous allons aussi ajouter les commandes énoncées en 3.1 au patron de conception Commande de la première version. De plus, nous ajouterons des classes de type **CommandeDefaireRefaire** prenant en compte la gestion de la fonctionnalité Defaire/Refaire.

Remarque :
Un axe d'amélioration de ce projet pourrait être l'implémentation de la suite de fibonacci pour la gestion des états. Ou encore mieux, l'implémentation d'un hybride des deux méthodes où passé une certaine taille de fichier le mini-éditeur changera de méthode pour privilégier la méthode de fibonacci.

3.3 - Scénarios

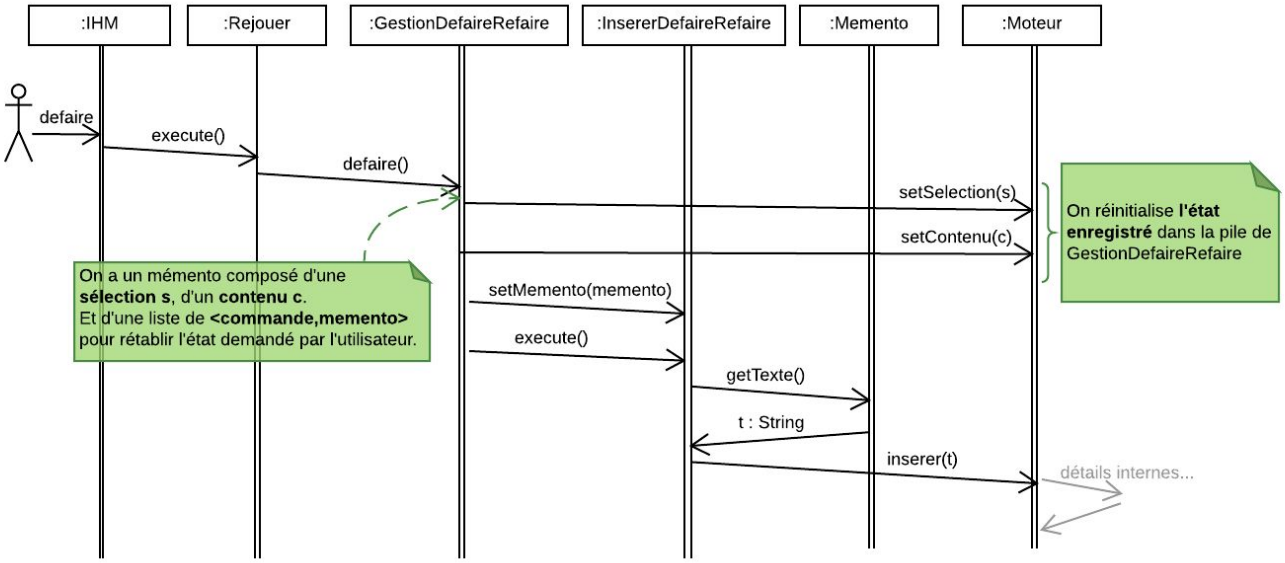

Figure 12 : Diagramme de séquence pour l'action de défaire en revenant à un état où il faut faire un Inserer pour aller dans l'état demandé par l'utilisateur

3.4 - Tests

Pour tester notre version 3 nous avons utilisé trois méthodes :

- Remplir le buffer à l'aide d'une suite de **n** commandes diverses. Puis à l'aide de la commande Défaire de la version 3 nous sommes retourné à l'état initial avec **n** commandes Défaire. Le test consistait à voir si le buffer et la sélection était dans le même état.

- La seconde méthode, inspirée de la première, consiste à voir qu'avec une suite de commandes avec un buffer non vide (et donc des commandes déjà exécutées) en exécutant une suite d'actions Défaire on retrouve bien l'état dans lequel on était. Le buffer n'étant pas vide cela complique le test.

- La troisième consiste à exécuter un scénario, le défaire (intégralement ou pas) puis refaire le scénario pour qu'on retrouve l'état dans lequel on était.

D'autres tests plus simple comme le test d'un défaire ou d'un refaire sur un buffer où aucune actions n'a encore été effectuées.

Ces tests nous ont permis d'identifier un problème toujours pas rétabli lorsque l'on applique la politique de stockage des états de buffer avec n > 1.

Conclusion

L'application des différents patrons de conception (Commande et Mémento) pour les différentes versions nous avons pu ajouter des fonctionnalités à notre mini-éditeur sans revenir (ou presque) sur le code déjà implémentées dans les versions antérieures.

Appréhender les différents patrons de conceptions nous a permis de mieux comprendre les rôles de chaque interface, chaque classe. De plus, l'utilisation des diagrammes de séquence nous a permis de . De cette façon nous avons pu factoriser notre code de façon modulaire et les tests nous permis d'identifier les reliquats que notre programme possède encore.

Si nous avions eu le temps, nous aurions voulu ajouter le patron de conception *Observer* sur l'IHM pour qu'elle soit notifiée de tout changement sur le buffer. L'interface homme-machine est un axe d'amélioration pour le projet pour lequel nous n'avions pas eu assez de temps car nous avons priorisé la conception des différents modules de ce projet.