Université Rennes 1

MASTER 1 Génie Logiciel

ACO 2016-2017

PROJET _ Mini Éditeur

Binôme :

MERZOUK Abdelghani 2.1

MERLET-THOMAZEAU Timothée 2.1

1. Introduction :

L'objectif de ce projet était de concevoir un éditeur de texte minimaliste en utilisant différents patrons de conception vu en cours et en TD. Ce rapport présente notre travail et nos choix de conception illustrés par de nombreux diagrammes en notation UML.

2. Présentation Globale :

Notre éditeur de texte minimaliste est uniquement doté des commandes de base : insertion de texte, sélection du texte, couper, coller et copier. L'implémentation se compose successivement de trois versions, chacune apportant de nouvelles fonctionnalités a la version qui la précède, ce qui a nécessité l'utilisation de nombreux patrons de conception (Command, Observer et Memento).

a) **La première version** ne permet l'utilisation que des commandes de base citées ci-dessus. Cette version sert de base à tout le projet. Elle comprend déjà un moteuréditeur,un pressepapiers ainsi que l'interface graphique que nous retrouvons dans toutes les autres versions.

b) **La deuxième version** implémente un système de MacroCommandes.
Une suite de commandes de base est enregistrée et qui pourra ainsi être rejouée.

c) Enfin **l'ultime version** ajoute les commande classique : Défaire et Refaire qu'on ne présente désormais plus.

V1_ Première version :

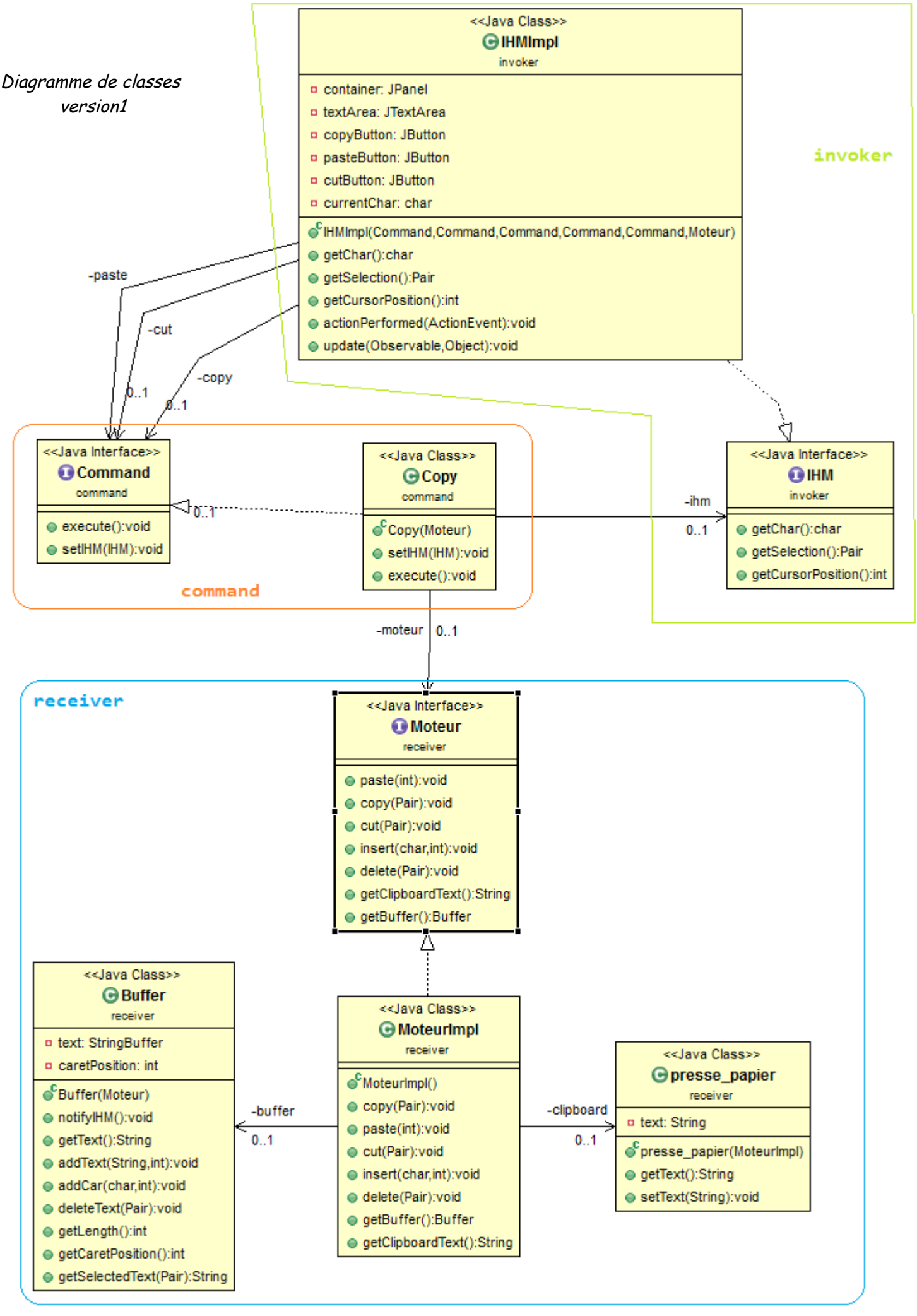
Le **Presse_papier** est très simple. Il sauvegarde le résultat des commande Copier et Couper, et le restitue lors de l'appel de la commande Coller.
Le Buffer contient l'état courant de notre mini éditeur. C'est à dire le texte

Le Moteur stocke les différentes informations telle que la sélection courante et les méthodes de bases permettant d'ajouter du texte, d'en supprimer, de retourner la sélection, de changer la sélection etc.

C'est le Moteur qui fait le lien entre le Pressepapier et le Buffer. Il gère l'appel les méthodes de ces deux entités de manière à obtenir l'effet désiré. (Par ex « Couper » : récupère la sélection dans le Buffer puis la supprime et enfin l'enregistre dans le Pressepapier),

Enfin, l'**IHM** gère l'affichage du mini éditeur ainsi que les interactions de l'utilisateur via le clavier ou la souris. Elle est muni d'une méthode `getCar()` qui permet a l'éditeur de connaître le dernier caractère entré par l'utilisateur, et dispose de 3 boutons: copier, couper et coller.

Diagramme de classes
version1



Le patron de conception Command:

Notre mini éditeur se repose sur la séparation maximale entre les différents composants. Ainsi, l'interface homme machine (IHM) et les autres classes (Moteur, Buffer et Pressepapier) sont dans deux packages différents. Par ailleurs, nous avons choisi d'implémenter le patron de conception Command ce qui nous permet de traiter toutes les commandes de base de manière générique.

Les commandes sont donc représentées par des classes qui implémentent l'interface Command. Le **Moteur** en est le « **receiver** » et **l'IHM** est « **l'invoker**. » Toutes les commandes sont munies d'une même méthode : `execute()`

Lors de l'appel d'une commande par l'utilisateur, l'IHM se contente donc d'appeler la méthode execute() de la commande correspondante. L'IHM et le Moteur sont ainsi bien séparés. Il est par conséquent aisé de faire des modifications sur l'un ou sur l'autre de manière indépendante.

Le diagramme de séquence ci-dessous explique l'enchaînement des opérations appelés lorsque l'utilisateur tape un caractère au clavier :

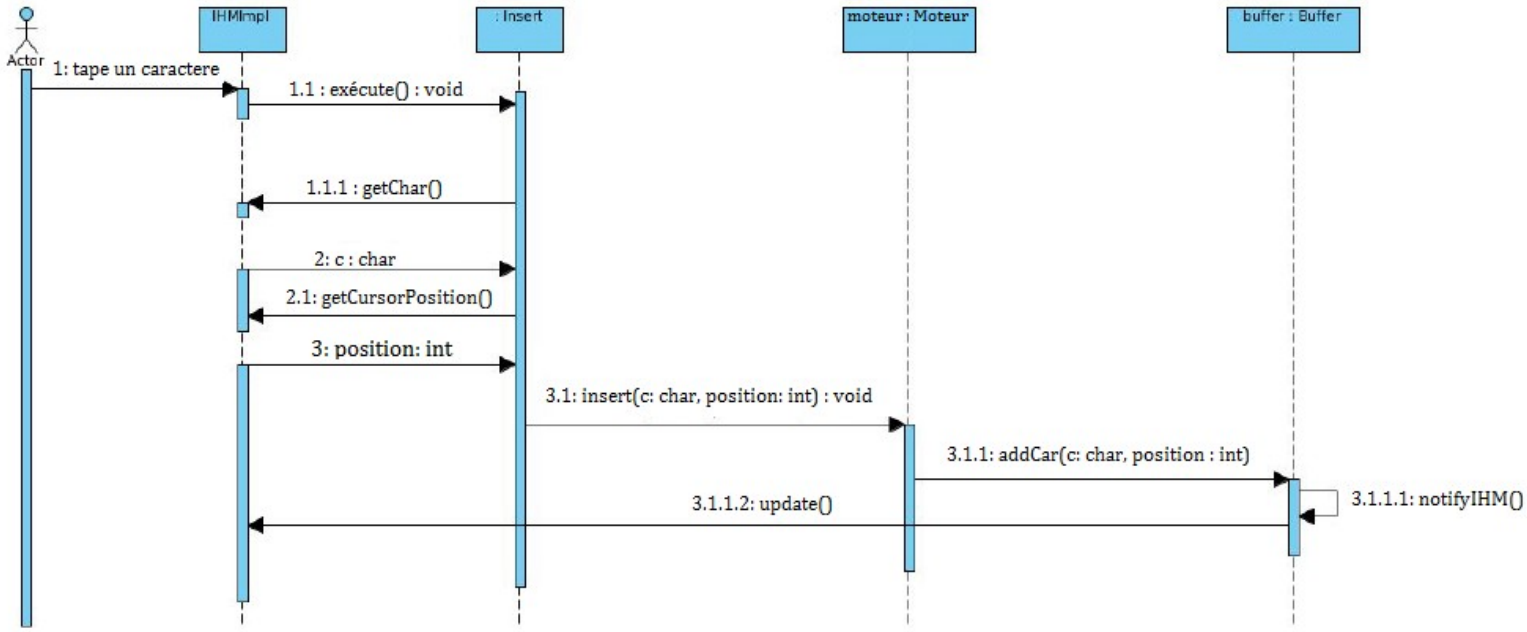


Diagramme de séquences lors d'une frappe au clavier

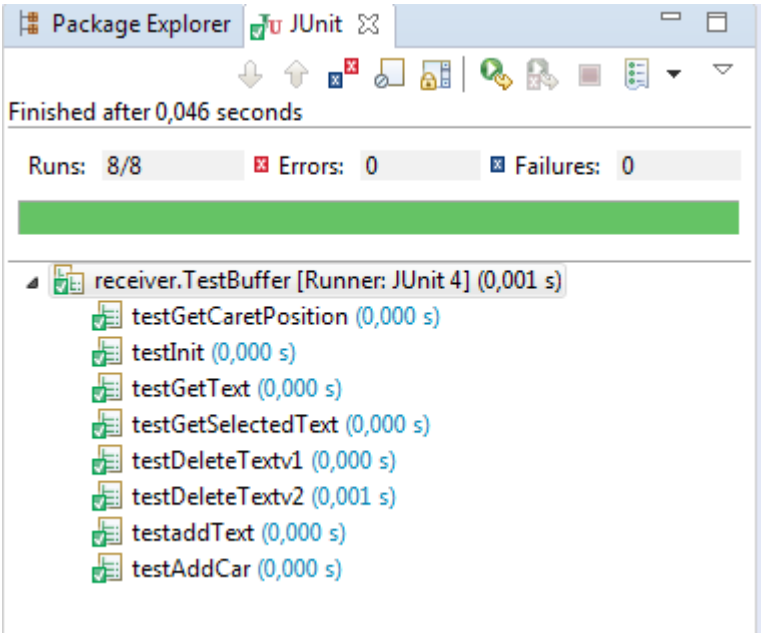
Le patron Observer :

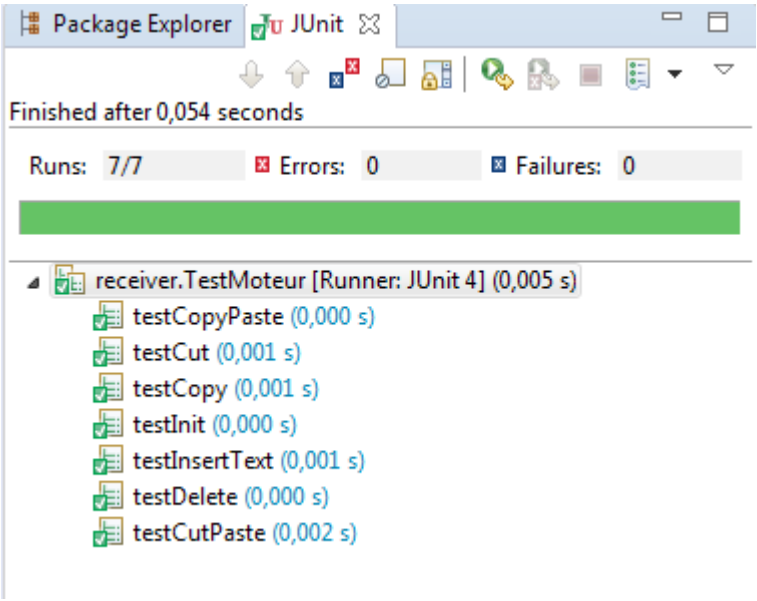
L'IHM devant afficher le texte contenue dans le Buffer, il faut qu'elle soit mise a jour à chaque changement de celuici. Pour cela nous avons utilisé le patron de conception ***Observer***.

Nous utilisons pour cela l'interface Observer de java qu'implémente l'IHM (observateur qui est notifié du nouvel etat du sujet). Cet observateur et ajouté au Buffer (sujet qui va changer d'etat) qui implémente l'interface Observable de

java. Ainsi avec la méthode `notify()` le Buffer signale a l'IHM qu'elle doit se mettre a jour.

Tests de la v1 (Buffer et moteur) :





V2_ la deuxième version /

L'ajout de macros nécessite l'utilisation d'un autre patron de conception qui sera « **Memento** », pour cela dans un package « memento », on a distribué les rôles comme suit :

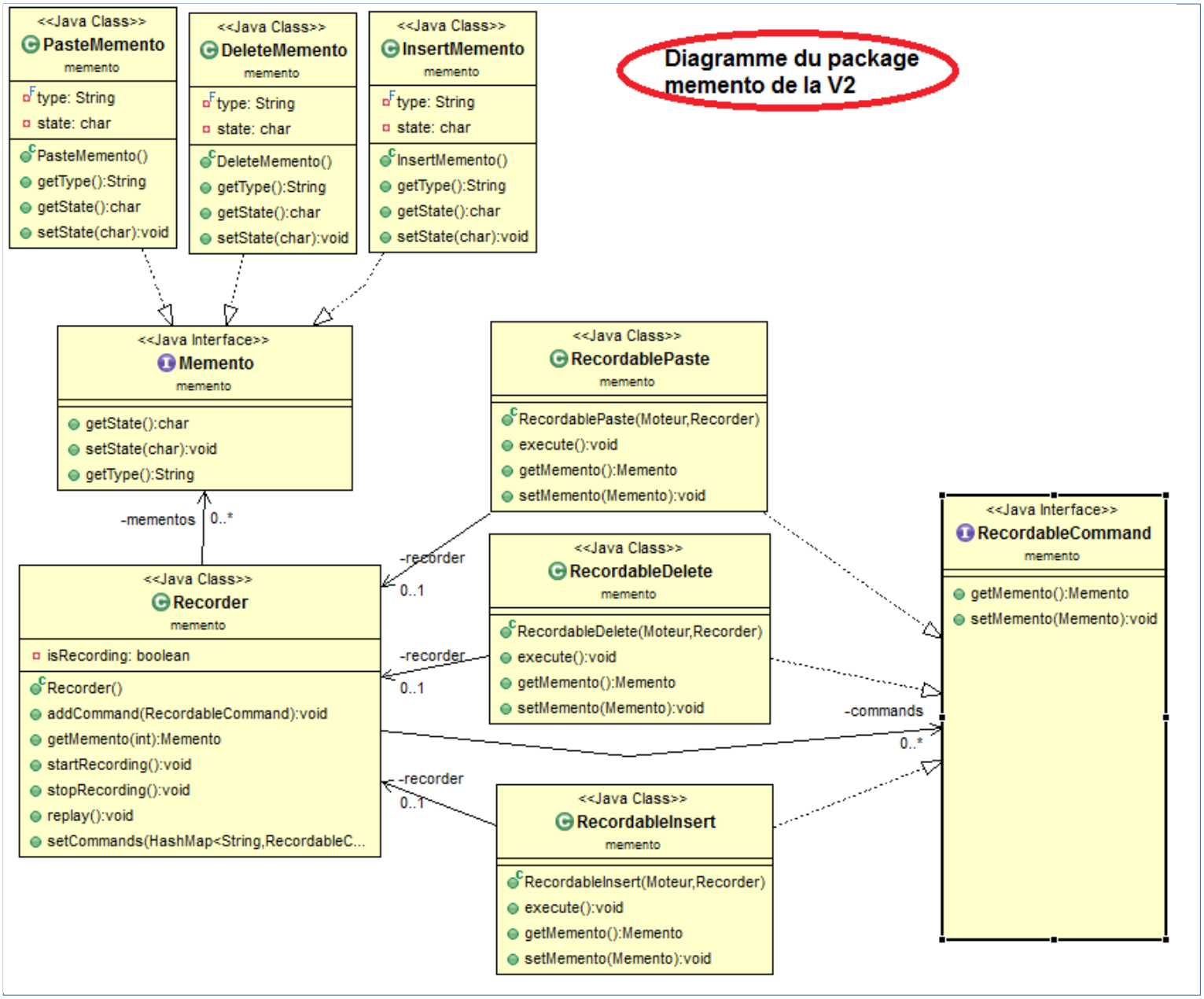
1. Memento : Conserve le nom de la commande enregistrée, et un état (dans notre cas c'est un caractère qui sera seulement nécessaire pour rejouer la commande Insert)

2. Enregistreur: joue le rôle du « **Caretaker** » dans le patron memento, il conserve les différents mementos enregistrés afin de les rejouer les commandes enregistrés après, et dispose aussi d'une variable isRecording afin de savoir si l'enregistrement a été activé ou pas.

3. Commandes enregistrables : qui sera notre « **Originator** » quand l'enregistrement est activé la commande demande a l'Enregistreur d'ajouter un Memento associé a cette commande dans la liste des Mementos

On a décidé de n'enregistrer que les commandes : insérer, coller et supprimer. Puisque les autres commandes (copier, couper) nécessitent plus d'informations et sont plus susceptibles de créer des bugs.

Afin de gérer les macros on a rajouté des boutons dans notre IHM :
• Start: pour commencer un enregistrement
• Stop : pour arrêter l'enregistrement
• Replay : afin de rejouer la séquence de commandes précédemment enregistrés



Tests de la v2 (Buffer et moteur) :

Finished after 0,054 seconds

Runs: 8/8 Errors: 0 Failures: 0

- receiver.TestBuffer [Runner: JUnit 4] (0,006 s)
  - testGetCaretPosition (0,000 s)
  - testInit (0,000 s)
  - testGetText (0,000 s)
  - testGetSelectedText (0,003 s)
  - testDeleteTextv1 (0,001 s)
  - testDeleteTextv2 (0,001 s)
  - testaddText (0,000 s)
  - testAddCar (0,001 s)

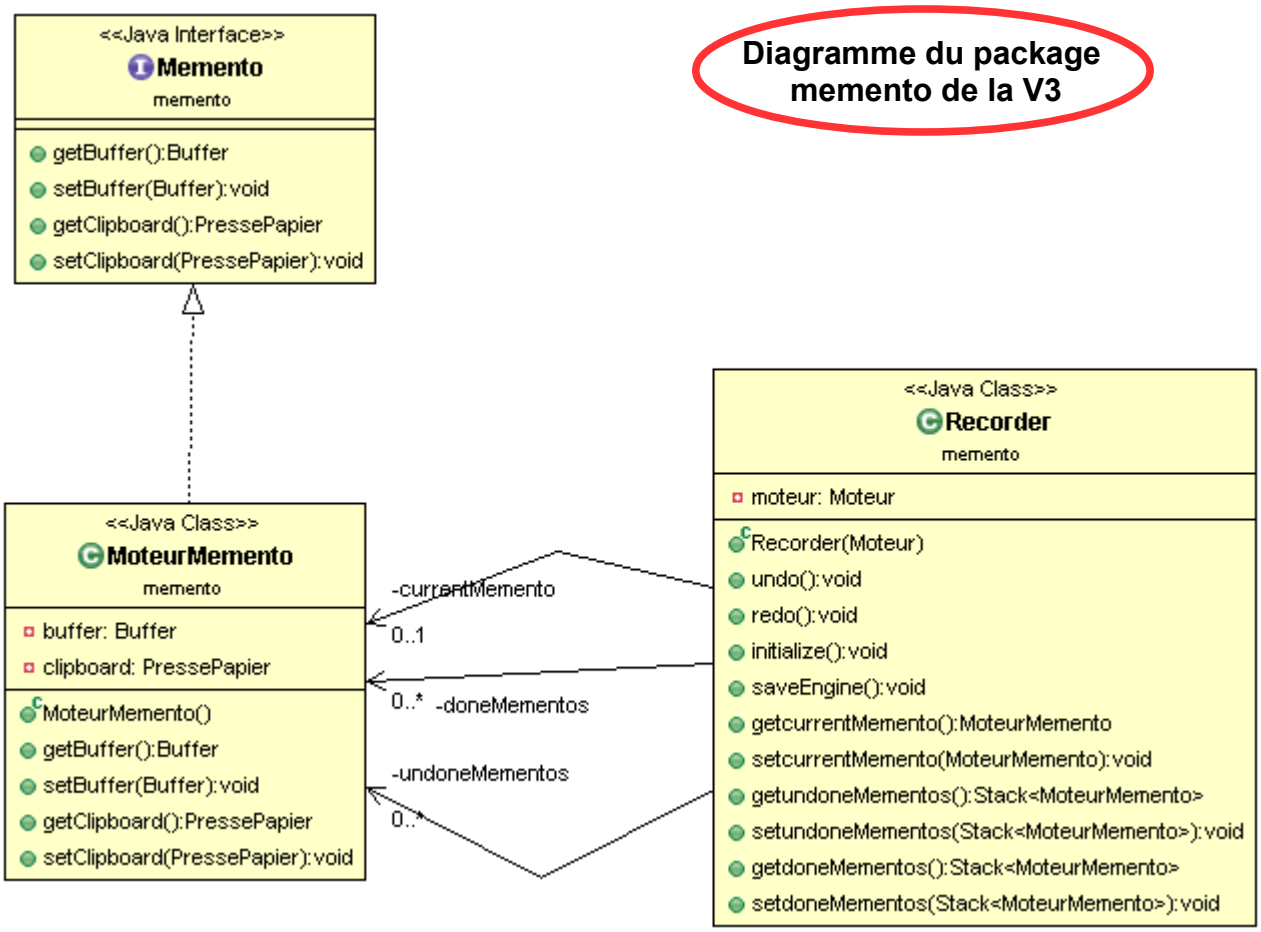
Finished after 0,085 seconds

Runs: 7/7 Errors: 0 Failures: 0

- receiver.TestMoteur [Runner: JUnit 4] (0,009 s)
  - testCopyPaste (0,008 s)
  - testCut (0,000 s)
  - testCopy (0,000 s)
  - testInit (0,000 s)
  - testInsertText (0,000 s)
  - testDelete (0,000 s)
  - testCutPaste (0,000 s)

V3_Troisième version /

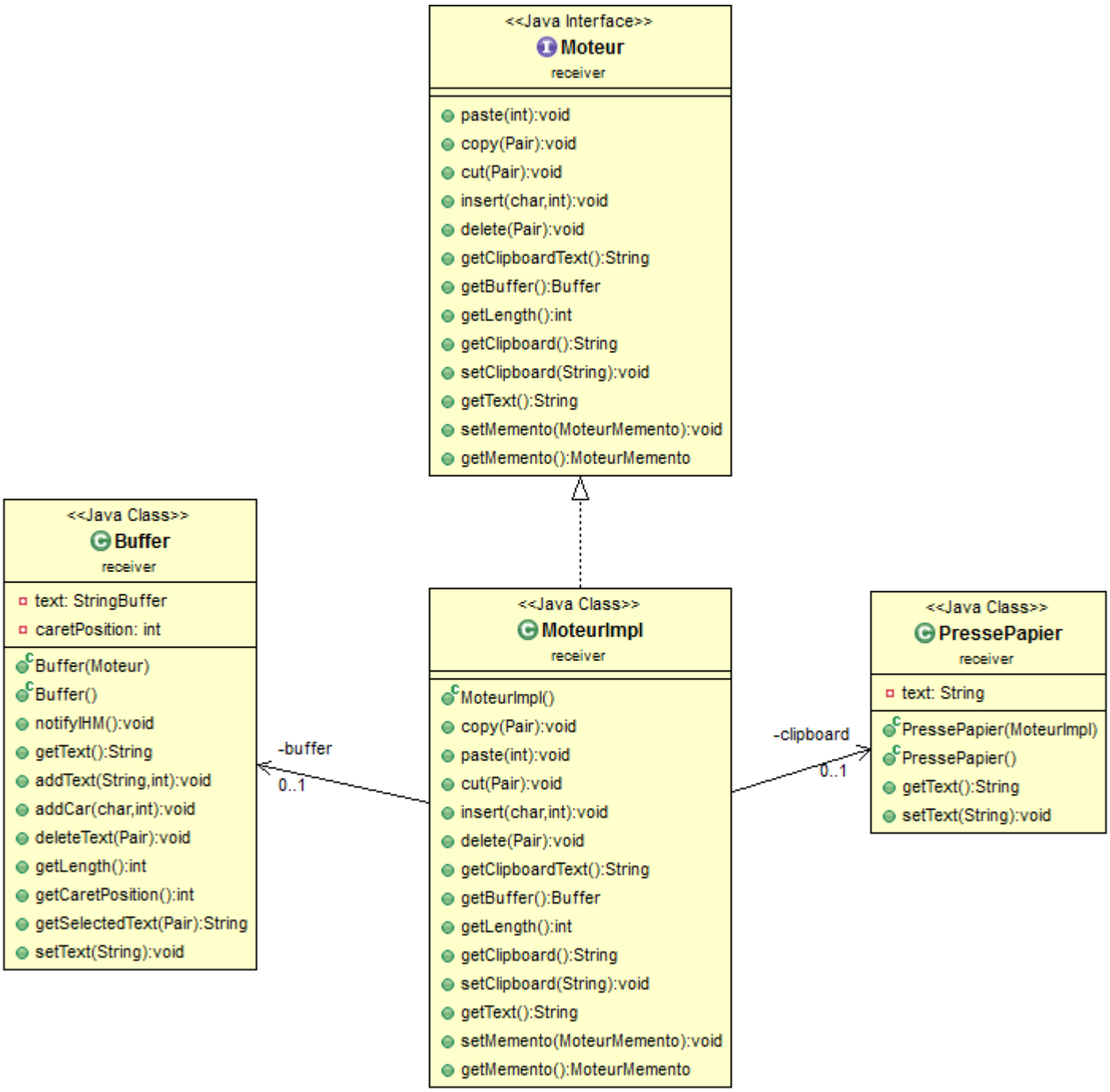
Dans cette version on va stocker l'état du Moteur avant chaque exécution d'une commande, comme le montre le diagramme ci dessous :



De plus on aura deux piles qui vont servir a stocker les MoteurMemento, a chaque fois qu'une commande est exécutée on enregistre dans la pile `doneMementos` l'état courant du moteur avant d'exécuter la commande. et par la suite quand on veut *défaire* cette commande on dépile le MoteurMemento stocké dans `doneMementos` et on rétablit ainsi l'état du buffer et du pressepapiers, ensuite on empilera le MoteurMemento dépilé dans la pile `undoneMementos`.

Pour refaire une commande, on va juste dépiler le MoteurMemento stocké dans la pile `undoneMementos`, et comme vu cidessus on rétablit l'état du buffer et du pressepapiers, et on empilera le MoteurMemento dépilé dans la pile `doneMementos`.

Le Moteur sera **l'Originator** du patron Memento et implémentera ainsi les méthodes : `setMemento()` et `getMemento()` comme le montre le diagramme de classes du package receiver ci-dessous:



Conclusion :

Ce projet fut l'occasion d'implémenter un mini éditeur simple et fonctionnel *from scratch* pour nous familiariser avec différents types de patrons de conception et également savoir écrire des tests et gérer des bugs.