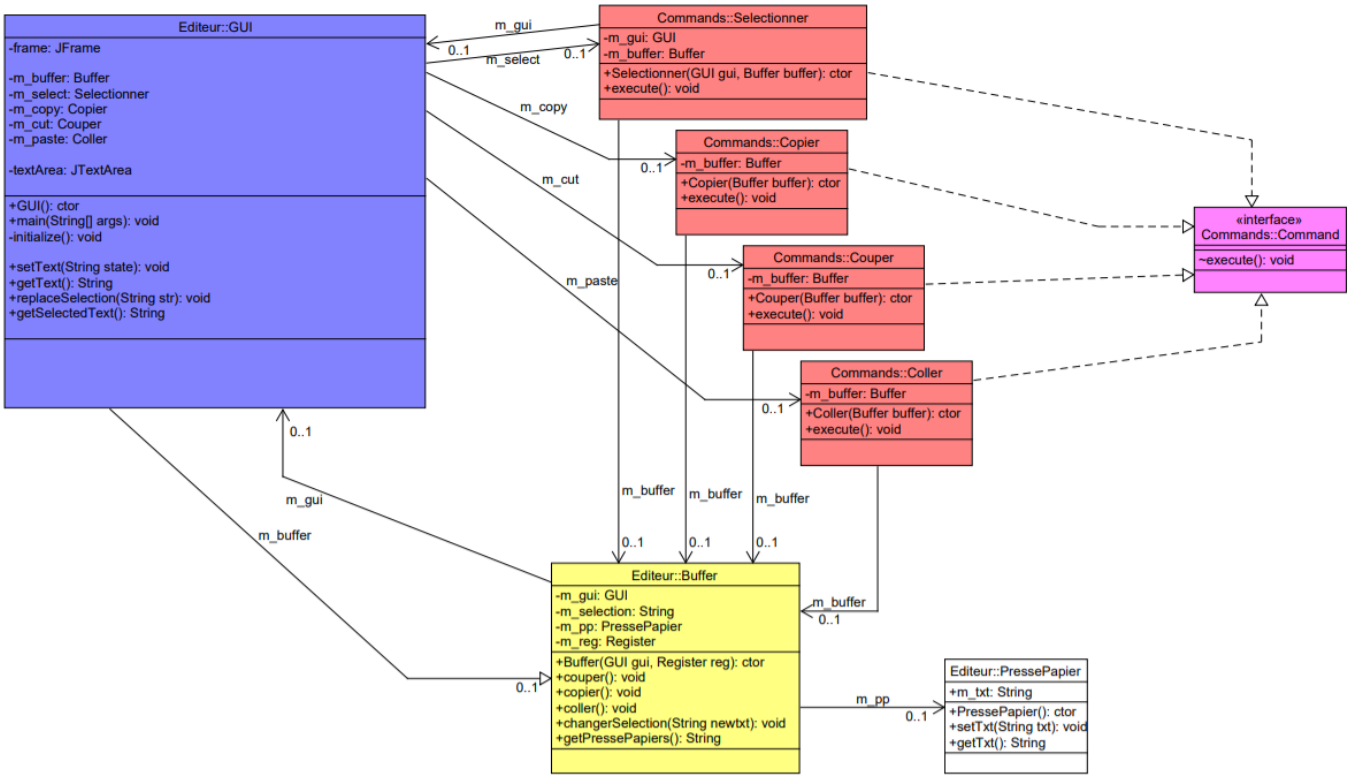
Compte rendu : projet MDI

Version 1

L'objectif de ce TP est de créer un éditeur de texte avec les diverses commandes qui lui sont associées. Pour cela nous aurons besoin d'une interface graphique ainsi qu'une zone de mémoire afin de stocker les valeurs souhaitées. Cependant l'interface graphique doit importer un nombre conséquent de méthodes liées à notre mémoire virtuelle afin de faire fonctionner correctement les différentes fonctionnalités de notre éditeur de texte. Pour éviter ce genre de problème nous utilisons un patron de conception nommé commande.

Il se décompose de la manière suivante : un "Invocateur" (notre interface graphique GUI), une interface "Commande" contenant toutes les "Commandes Concrètes". Ces commandes appellent ensuite notre buffer qui jouera le rôle de "Récepteur". Le diagramme UML est représenté ci-dessous :



Commandes Copier / Couper / Coller / Sélectionner

Comme vu dans le design pattern, les commandes serviront à communiquer avec notre buffer sur l'action à réaliser. Ces actions sont représentées par la fonction execute() de chaque commandes dont chacune des classes implémentant l'interface Command dispose. Cela nous permet ainsi d'éviter d'implémenter simplement nos commandes.

Le fonctionnement de ces commandes restent relativement basique. Lorsque l'utilisateur clique dans le menu ou effectue le raccourci associé à une commande nous appelons la fonction execute() de celle-ci. Par exemple pour la commande "Selectionner", la fonction execute() nous permet d'enregistrer la sélection dans un attribut "m_selection" dans le buffer à chaque fois que la sélection change dans l'IHM (utilisation de Listener). Les commandes "Copier/Couper" permettent d'enregistrer dans le presse papier du buffer, la sélection récupérer grâce à la commande "Selectionner" présenté précédemment, en supprimant la sélection actuel dans la JTextArea de l'IHM si nous effectuons un "Couper".

Dans le cas de la commande “Coller”, nous accédons au texte du presse-papier afin de coller le texte à l’endroit souhaité dans la JTextArea grâce à la fonction “replaceSelection(String str)” implémentée dans l’IHM.

Test sur la version 1

Afin de valider notre première implémentation de notre éditeur de texte nous avons créé un fichier JUnit, qui teste plusieurs cas possible d’utilisation. Nos premiers tests nous permettent de vérifier la bonne fonctionnalité de commande de base telle que “Sélection”, dans ce cas nous testons que notre attribut “m_selection” de notre Buffer est bien remplacé par la sélection dans la JTextArea lorsque l’on appelle cette commande ou qu’il ne se passe rien si aucun texte n’a été sélectionné. De plus, nous testons de la même manière le contenu du texte enregistré dans le presse-papier avec la commande “Copier” et “Couper”. Avec la commande “Couper”, nous testons également le contenu de la JTextArea en fonction de la sélection qu’elle soit vide ou non.

Pour valider la commande “Coller”, nous testons le cas lorsque le curseur sélectionne un texte, dans ce cas nous regardons si le contenu de la sélection est bien remplacé par le contenu du presse-papier. Si le curseur est placé à une position sans rien sélectionner, alors on test l’ajout simple du contenu du presse-papier sans modification du texte initiale.

Chaque test est effectué en simulant l’appui sur les touches afin d’utiliser les raccourcies associés aux commandes, et en passant par le menu de l’IHM.

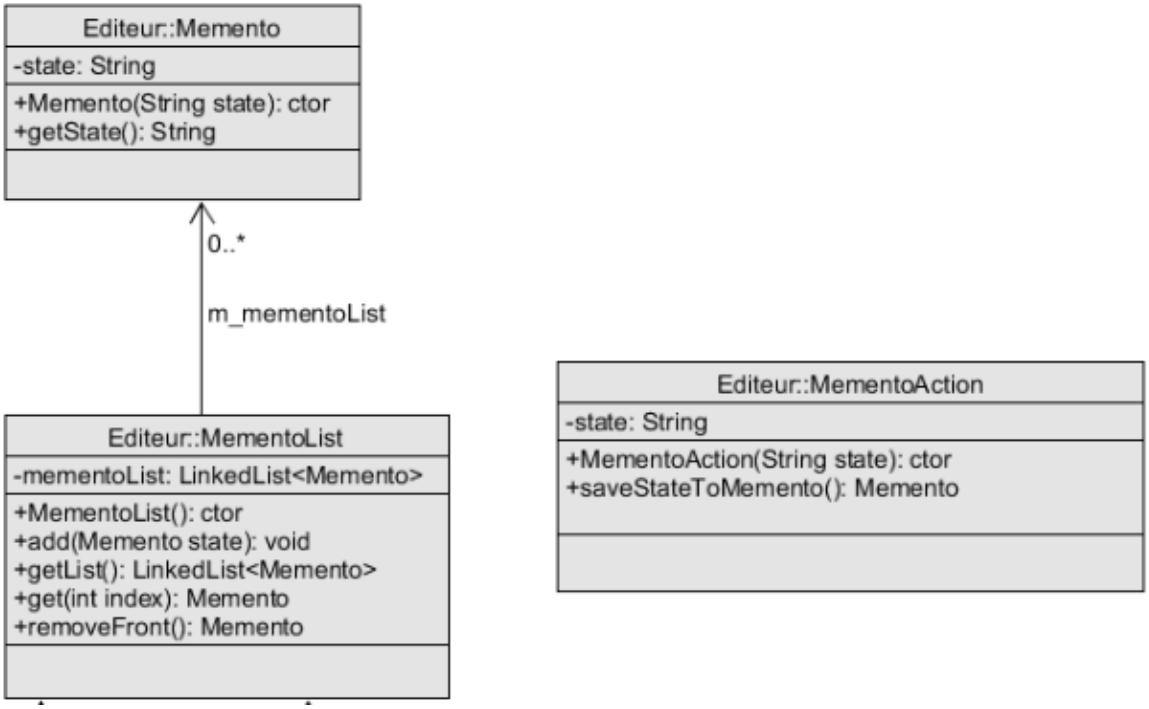
Version 2

La version 2 est une version améliorée de notre éditeur de texte puisqu’il est désormais possible de revenir en arrière avec la commande Undo et de faire un retour vers l’avant avec Redo. Il devient aussi possible d’enregistrer jusqu’à 3 macros afin qu’ils puissent contenir une suite de commande et qu’il soit possible de les exécuter en actionnant la macro associée.

Commandes Undo / Redo :

Afin de conceptualiser les commandes Undo (Ctrl + Z) et Redo (Ctrl + Y) nous avons décidé d’utiliser le design pattern Memento. Celui-ci nous permettra de restaurer un objet dans son état précédent tout en respectant le principe d’encapsulation (Modifier les structures de données sans impact sur l’interface).

Nous utilisons 3 classes différentes afin de correspondre au design pattern : MementoAction qui peut être associée à la classe appelé “Originator” du pattern, “Memento”, et MementoList correspondant à la classe “CareTaker” du pattern.



Design pattern Memento

Dans notre cas, nous voulons implémenter un retour en arrière dans notre éditeur de texte. Nous avons donc décidé d'enregistrer le texte présent dans la JTextArea dans un String dans la classe Memento. Chaque instance de Memento sera par la suite sauvegarder dans une liste contenue dans une instance de type MementoList. Ainsi, il sera possible de sauvegarder plusieurs états dans un ordre donnée. Pour réaliser ces sauvegardes nous utiliserons notre classe MementoAction qui est composée d'une fonction saveStateToMemento() permettant l'ajout d'un Memento dans notre MementoList.

La commande redo fonctionne de la même manière mis à part que la liste des Mementos se remplie lorsque l'on retire un élément de la liste des Memento de undo.

Nous avons décidé d'implémenter les commandes Undo et Redo en utilisant une liste pour chaque Undo possible et une liste pour chaque Redo possible. Celles-ci seront stockées dans la classe Register composée de fonctions permettant d'initialiser et de clear la liste ainsi que des fonctions pour ajouter des éléments dans la liste correspondante.

Le système est de nouveau semblable au design pattern de Command. Seul le récepteur change (Register). Plus de détails sont donnés par la suite.

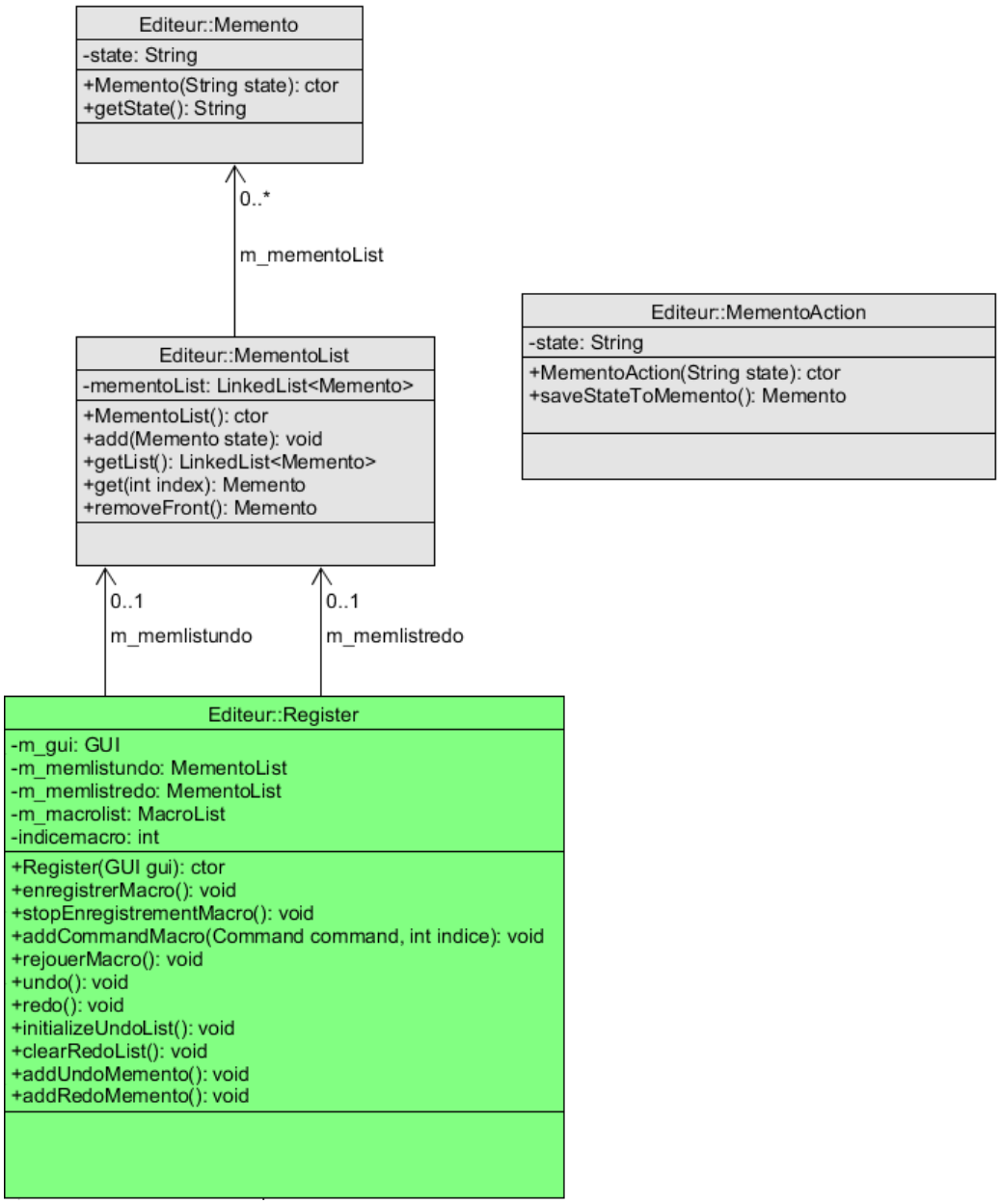


Diagramme UML du design Pattern Memento avec un Register

Tests effectués sur Undo / Redo

Afin de vérifier la bonne fonctionnalité des commandes Undo et Redo, nous avons mis en place des fonctions permettant de tester l'utilisation de la commande Undo dans le cas où il n'y a pas de retour en arrière possible, ainsi que lorsque l'on a effectué différent type de commande tel que "Couper", "Coller", ou "Redo" qui ont la propriété d'enregistrer l'état du texte avant l'exécution de la commande. De plus, nous testons Undo après l'utilisation de la touche effacement arrière qui enregistre avant l'exécution de la touche comme les commandes précédentes et de la barre d'espace, de la touche entrée qui enregistre cette fois ci après l'exécution de la touche.

Afin de vérifier la commande "Redo", l'exécution est testé de la même manière que la commande "Undo", après l'appel des commandes "Couper", "Coller", "Undo", de la touche espace, entrée et effacement arrière. De même, lorsqu'il est impossible d'avancer dans la liste des Redo. De plus, la commande "Redo" doit être testé après être retourné en arrière pour voir si la liste de Redo a bien été réinitialisé.

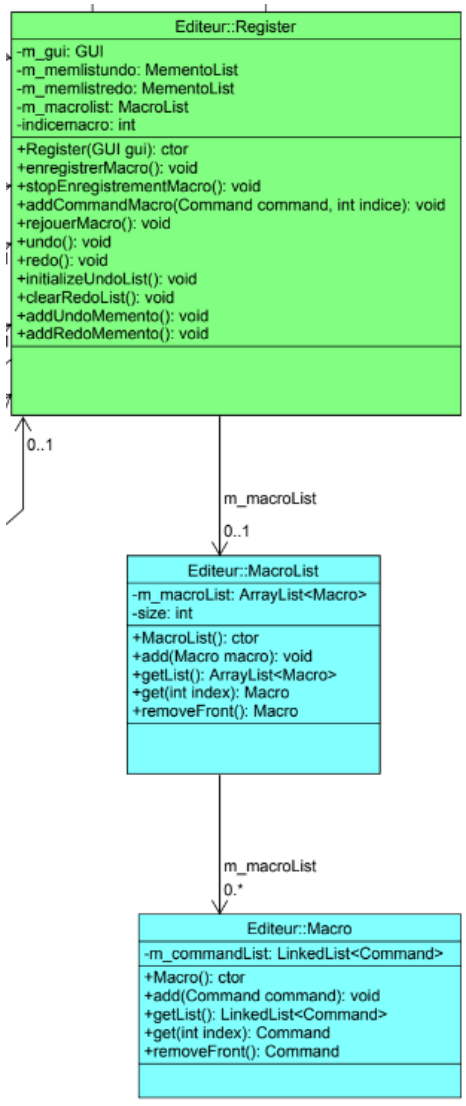
Ces commandes sont aussi validées en utilisant à la fois les raccourcies claviers et en passant par le menu Edit de l'IHM.

Enregistrer des Macros

Dans notre éditeur de texte, il est possible de d'enregistrer 3 macros différentes (Appuyez sur F1 pour commencer l'enregistrement du Script 1, F2 pour 2 et F3 pour 3). Celui-ci enregistrera les commandes tapées par l'utilisateur. Une fois cette étape terminée appuyez de nouveau sur F1, F2 ou F3 afin de terminer l'enregistrement. Il sera alors possible d'exécuter le Script depuis le menu ou en appuyant sur Ctrl+1, Ctrl+2 ou Ctrl+3. Les commandes enregistrables sont Copier, Coller, Couper, Undo, Redo.

Dans cette 2ème version, afin d'implémenter nos nouvelles commandes "Undo", "Redo", ainsi que les commandes liées aux macros, nous repartons de notre design pattern command initiale en rajoutant un nouveau "Recepteur" appelé "Register" en plus du "Buffer". Les commandes de notre 2ème version seront les suivantes : "Undo" / "Redo" / "RejouerMacro" / "EnregisterMacro" / "StopEnregistrementMacro" (couleur orange dans l'UML). Ce "Register" prend en attribut des listes de "Memento", une pour le Undo, une pour le Redo ainsi qu'une "MacroList". Il représente notre zone mémoire d'enregistrement pour les commandes de la version 2.

Pour les macros nous avons utilisé un design pattern semblable au "Memento". Nous disposons d'une classe "Macro" prenant comme attribut une liste de "Command", puis nous avons une classe "MacroList" prenant une ArrayList, composée de nos 3 différentes macros enregistrables, en attribut (couleur cyan dans l'UML). Cette fois-ci nous avons utilisé 2 classes différentes sans passer par une classe "Originator" habituellement présent dans ce design pattern.



UML Macro

Ainsi le diagramme UML final prend la forme suivante (Pour un meilleur aperçu, consulter le pdf)

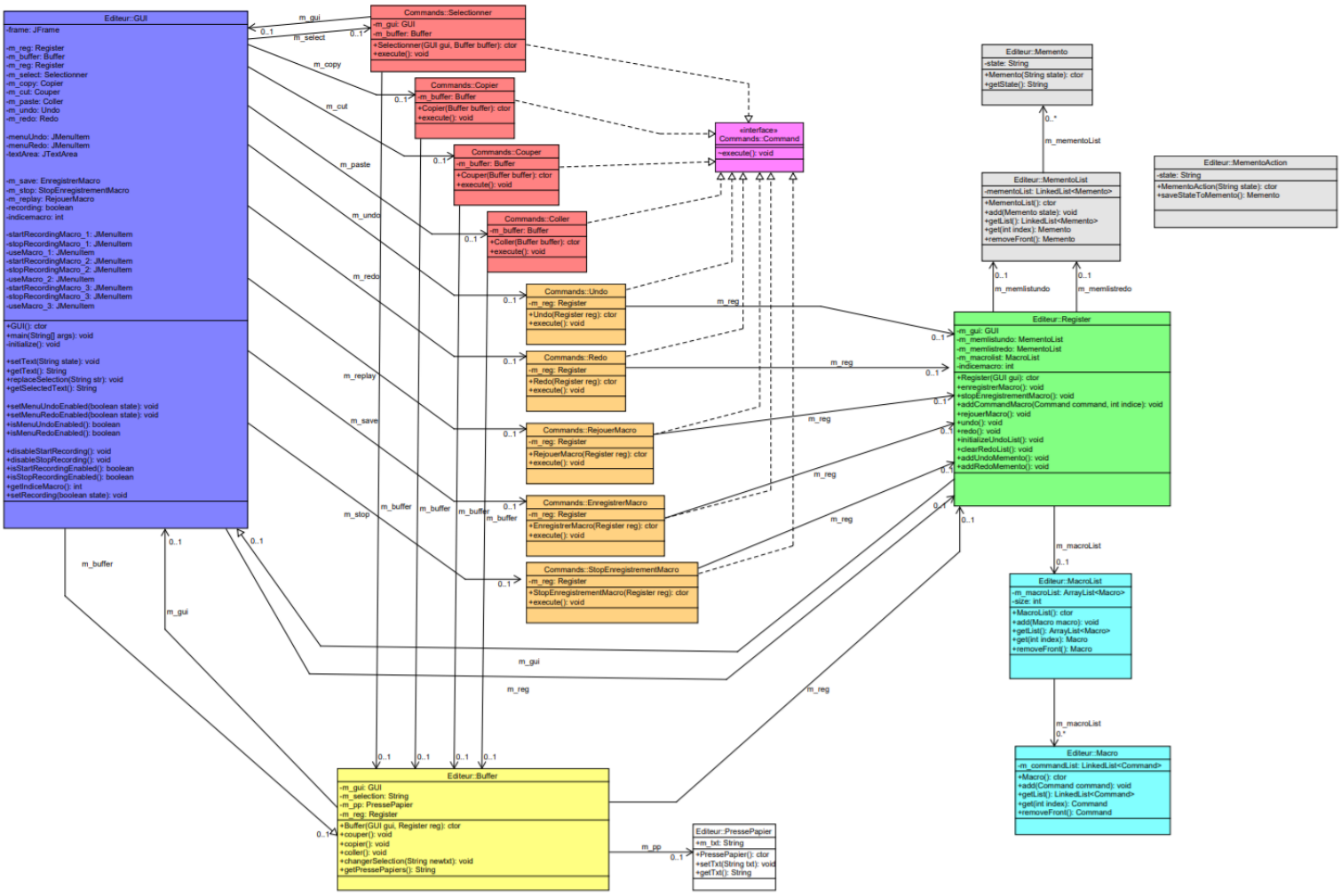


Diagramme UML final

Test des Macros :

Enfin, pour tester notre dernière version de notre programme implémentant les commandes liées aux macros, qui sont : “RejouerMacro”, “EnregisterMacro” et “StopEnregistrementMacro”, nous avons rajouté quelque fonctions supplémentaires.

Afin, de tester notre commande “EnregistrerMacro”, nous simulons l’appui des touches correspondantes à l’enregistrement/arrêt (F1,F2,F3) ainsi que l’appui par le menu des boutons Enregistrement et Arrêt associés à chaque macro. De plus, nous testons l'exécution des commandes de démarrage d'enregistrement après avoir effectué cette même commande afin de vérifier qu’il ne peut pas y avoir d'interférence entre enregistrement. Nous testons également l’enregistrement sur une macro déjà existante pour vérifier le remplacement de la macro.

Ensuite, pour tester l’arrêt d’une macro “StopEnregistrementMacro”, nous testons l'exécution de la commande sur toutes les macros et seul l'arrêt de la macro correspondant devrait fonctionner. Cette commande est aussi testée lorsqu’aucun démarrage n’a pas été débuté.

La commande d'exécution “RejouerMacro” est vérifiée en exécutant la commande lorsqu’aucun enregistrement est en cours, si la macro n’a pas encore été enregistrée rien ne devrait se produire. Si la commande s'exécute la fonction teste le texte de la JTextArea de notre IHM afin de vérifier le bon fonctionnement de la macro en fonction de l’enregistrement effectué. Un test d'exécution d’une macro lors de l’enregistrement est aussi effectué et permet de vérifier qu’aucune macro ne peut être effectuée tant que nous sommes en enregistrement.

Conclusion: En conclusion, nous avons créé un éditeur de texte avec une interface graphique utilisable avec clavier ou avec souris. L'emploie de designs patterns existants nous a facilité le travail et nous a permis d'obtenir un code correctement structuré. Le code est opérationnel et est entièrement testé par un test en JUnit complet. Néanmoins une meilleure structuration de nos tests auraient pu être effectué afin de faciliter le développeur en cas de problème.