Outils et méthodes pour le développement
TP – Conception Orientée-Objet - Mini éditeur

Molinier Camille

28/09/2022

J'atteste que ce travail est original, qu'il indique de façon appropriée tous les emprunts, et qu'il fait référence de façon appropriée à chaque source utilisée.

Table des matières

1 Introduction

Le but de ce TP est de concevoir un mini éditeur de texte. Cet éditeur doit permettre d'effectuer les commandes classiques d'un éditeur de texte classique (comme le bloc note natif de Windows par exemple). On a vu dans le rapport de conception tous les aspects de réflexions autour de l'architecture du projet. Ce rapport traitera donc de la partie implémentation et choix techniques pour concrétiser la conception. On verra comment chaque version est faite et comment on est passé de la première à la seconde.

2 Version 1

Cette première version fait intervenir des commandes de bases. Elle sera aussi la version qui contiendra le premier squelette du projet. Afin de procéder de façon sereine, on commence par implémenter le diagramme de classe puis on procède fonctionnalité par fonctionnalité

2.1 Implémentation du diagramme de classes

Pour commencer l'implémentation, on commence par traduire le diagramme de classes en fichier de classes.
On commence par créer toutes les interfaces avec leurs déclarations de méthodes, car elles sont au cœur de la structure du projet. On crée ensuite les fichiers de classes dont on aura besoin en indiquant bien l'interface qu'elles implémentent. Le premier avantage des interfaces se révèle alors être le gain de temps à créer toutes les commandes, car les méthodes se déclarent toutes seules dès que l'on indique qu'elles implémentent de l'interface *Command*.
Maintenant que l'architecture du projet est en place, on peut créer les liens entre les classes (en déclarant les attributs) puis commencer à implémenter nos fonctionnalités.

2.2 Implémentation de l'interface graphique

Pour l'implémentation de l'interface graphique, on a choisi d'utiliser la librairie Swing. La classe *MyGUI* étend donc la classe *JFrame* et implémente l'interface *GUI*. Pour permettre à notre application de reconnaitre les saisies clavier ainsi que les clics sur des boutons, on ajoute l'implémentation des interfaces *KeyListener* et *ActionListener*.

La classe doit donc implémenter les fonctions *update()*, *actionPerformed(ActionEvent e)*, *update()* et *keyPressed(KeyEvent e)*. Normalement, l'interface *KeyListener* vient avec les fonctions de relâchement de touche et d'activation de touche, mais celles-ci ne seront pas utiles à notre projet, donc leur implémentation restera vide.

Pour toute la partie concernant la JFrame, on utilise principalement un JTexterea dont on appelle la fonction *setEditable(false)* afin de ne pas profiter de ses fonctionnalités (ce qui rendraient inutile tout le code que l'on pourrait écrire) tout en ayant un conteneur graphique pour le texte.

La fonction *update()* permet de mettre à jour l'affichage à partir du contenu du buffer, vérifier que les curseurs ne sont pas en dehors du buffer, auquel cas, ils sont repositionnés, puis modifie l'affichage dans les labels indiquant les positions des curseurs.

2.3 Implémentation des fonctionnalités d'insertion et de suppression

Les premières fonctionnalités que l'on va implémenter sont l'insertion et la suppression de texte.

Pour insérer du texte, on implémente dans la GUI la fonction *keyPressed(KeyEvent e)* pour reconnaître les touches qui correspondent à du texte (les caractères de l'alphabet, les numéros et certains caractères spéciaux). On appelle la fonction *setCommand(Command c)* de l'*Invoker* dans la GUI avec une nouvelle commande de type *Insert* qui prends la touche en paramètre. Ensuite, on demande ensuite à l'*Invoker* d'exécuter la commande, celle-ci va appeler la fonction *insert(String c, int position)* de son *Engine* qui va demander au *Buffer* d'ajouter le contenu à la position indiquée. De ce fait, lorsque la GUI viendra appeler la fonction *update()*, elle récupèrera le contenu modifié du *Buffer* et afficher le caractère inséré.

Pour supprimer du texte, c'est quasiment le même cheminement. La GUI détecte l'appui sur la touche DEL dans la fonction *keyPressed(KeyEvent e)*, ce qui d'éclanche l'ajout d'une commande de type*Delete* dans l'*Invoker*. Lorsque cette commande est exécutée, elle demande à son *Engine* de supprimer le caractère à la position souhaitée. L'*Engine* appelle alors la fonction *deleteContent(int start, int stop)* du *Buffer* ce qui modifie son contenu.

2.4 Implémentation de la fonctionnalité copier/couper et coller/remplacer

Maintenant que l'on peut insérer et supprimer du texte, on peut ajouter la fonction pour copier, couper et coller du texte. Ces trois nouvelles fonctionnalités font intervenir un nouvel élément : le deuxième curseur. Celui-ci permet de définir une position relative au premier curseur et donc de définir une zone de sélection.

La commande de copie utilise directement les positions des deux curseurs. Lorsque la fonction *actionPerformed(ActionEvent e)* détecte l'appui sur le bouton de copie, une nouvelle commande *Copy* est donnée à l'*Invoker* avec les positions des curseurs en paramètre. Cette commande va appeler la fonction *copy(int start, int stop)* de l'*Engine* qui va venir récupérer le texte du *Buffer* entre les deux bornes puis le stocker dans le *Clipborad*
Pour la commande *Cut*, le principe est exactement le même sauf qu'une fois le texte dans le *Clipboard*, celui-ci est supprimé de la même façon que la commande *Delete*.

Pour la commande de collage du texte, on attend l'appui sur le bouton "paste" avec la fonction *actionPerformed(ActionEvent e)*. Si le bouton est activé, il déclenche l'insertion d'une nouvelle commande de type *Paste* dans l'*Invoker*. Lorsque cette commande est exécutée, elle vient récupérer le contenu du *Clipboard* puis l'insère dans le *Buffer* de la même façon que l'on insère du texte avec la commande *Insert*.

La dernière fonctionnalité implémentée dans cette première version est le remplacement d'une sélection. Le principe de fonctionnement est le même que pour la fonction de collage sauf qu'avant d'insérer le contenu du *Clipboard*, la commande *Replace* va supprimer le texte. C'est un peu comme un enchainement d'une fonction *Cut* suivi d'une fonction *Paste*.

Le choix entre uen commande *Paste* ou *Replace* se fait automatiquement, la *GUI* regarde simplement si les deux curseurs sont à la même position pour *Paste*, sinon il appelle une commande *Replace*.

3 Version 2

Cette deuxième version permet d'enregistrer des suites de commandes utilisateurs et de les rejouer ainsi que de pouvoir annuler ou refaire des commandes. De nouvelles classes vont apparaître notamment avec l'arrivée du patron de conception Memento

3.1 Implémentation des nouveautés du diagramme de classe

Pour commencer cette nouvelle version, on commence par ajouter toutes les nouveautés du diagramme de classes. On implémente toutes les nouvelles interfaces puis on ajoute leurs implémentations respectives. Enfin, on ajoute les nouvelles implémentations de l'interface commande : *Undo*, *Redo*, *Script* et *Load*.

Pour l'implémentation du patron Memento, on commence par implémenter la classe *Snapshot* qui aura en charge de contenir les diverses informations d'une sauvegarde précise. On choisit donc d'y stocker l'état du *Buffer* ainsi que la dernière commande entrée sous forme de liste de String (List(<nom commande>, <param1>, <param2>, ...). On définit aussi une méthode pour comparer les objets de type *Memento*, car on sait déjà que des comparaisons seront nécessaires lors des scripts (lorsque l'on voudra récupérer une sous-liste de *Memento* à partir d'un point de départ).

On passe ensuite à l'implémentation du *CareTaker* avec la classe *StackCareTaker*. Comme son nom l'indique, on a choisi l'utilisation d'une pile pour implémenter la sauvegarde des commandes, car ce qui nous intéressera sera de récupérer seulement le dernier état du système ou de revenir en avant. C'est pourquoi la classe *StackCareTaker* possède deux piles, la première gère les commandes entrées et la deuxième les commandes annulées. Au niveau des méthodes, on y trouve de quoi ajouter un *Memento*, récupérer le dernier *Memento* en le supprimant de la pile principale puis en le stockant dans la pile de backup, consulter le *Memento* au-dessus de la pile sans le supprimer, de rétablir le dernier *Memento* annulé ainsi qu'une méthode pour récupérer une liste à partir d'un *Memento* dans la pile.

Il reste à implémenter l'*Originator*, celui-ci possède trois méthodes. La première permet de faire une sauvegarde d'un nouveau *Memento*, la seconde permet de retourner le dernier *Memento* annulé et la troisième permet de rétablir dans le *StackCareTaker* le dernier *Memento* annulé.

Maintenant que le nouveau patron est maintenant fonctionnel, on fait une petite modification des implémentations de *Command* pour qu'elles effectuent une sauvegarde après exécution. On peut désormais implémenter les nouvelles fonctionnalités.

3.2 Implémentation des fonctionnalités Undo/Redo

Pour ces premières nouvelles fonctionnalités, on doit pouvoir annuler une commande ou revenir sur l'état avant annulation. On va donc utiliser les mécaniques du nouveau patron pour implémenter ses fonctionnalités.

Comme vu dans le rapport de conception (section 3.2), le fonctionnement de la commande *Undo* est assez basique. Lors de son exécution, la commande appelle la méthode *restore()* de l'*Originator* et récupère donc un objet de type *Memento*. On vient ensuite demander au *Recieiver* d'effacer tout le contenu du *Buffer* puis d'y insérer le contenu du *Memento*. De cette façon, lorsque la *GUI* mettra à jour son affichage, l'utilisateur verra un retour en arrière, même si en réalité tout le texte est remplacé. À noter que la commande *Undo* est la seule à ne pas être sauvegardée, ceci est dû au fait que si on sauvegarde une fonction *Undo* avec le fonctionnement actuel, elle occupera le sommet de la pile du *StacktCareTaker* en permanence et donc il sera impossible de revenir en arrière plus d'une fois.

Pour la commande *Redo*, le fonctionnement est sensiblement le même. La commande va appeler la méthode *respawn()* de l'*Originator* qui va demander au *CareTaker* de rétablir la commande sur le sommet de sa pile de backup puis retourner le nouveau *Memento* en sommet de pile principale. La commande va ensuite demander au *Reciever* de supprimer le contenu du *Buffer* puis y insérer le contenu sauvegardé dans l'objet *Memento*. Cette commande peut être sauvegardée dans la pile du *StackCareTaker* car elle appelle la pile de backup et donc n'occupera pas le sommet en permanence.

3.3 Implémentation du système de scripts

Maintenant que toutes les commandes sont implémentées, on peut réaliser le système de scripts. Pour stocker les suites de commandes, on aurait pu choisir de créer un objet dédié uniquement à la lecture/écriture. Mais, cela voudrait dire que lors de la fermeture du programme, les scripts seraient perdus. Donc, il a été décider de se servir du système de fichiers pour stocker les suites de commandes dans des fichiers sans extension.

La commande *Script* à un comportement particulier, ce qui lui vaut une implémentation sous forme de singleton. Lors de son premier appel, elle prépare le script en appelant la méthode *showMemento()* du *CareTaker*. Cette méthode va retourner le *Memento* courant qui sera stocké comme point de départ du script. Lors du second appel, la commande demande au *CareTaker* la liste des *Memento* à partir du point établi lors du précédent appel avec la méthode *getMementoList(memento)*. Cette liste est ensuite sauvegardée dans un fichier.

La sauvegarde dans un fichier se fait en plusieurs étapes. Tout d'abord, un fichier est ouvert (ou crée s'il n'existe pas) dans le dossier où sont rangés les scripts. On crée ensuite un objet *Writer* qui permettra d'écrire dans le fichier. Enfin, on parcourt la liste des commandes puis on les inscrit une à une dans le fichier. La seule exception est lorsque l'on veut enregistrer une commande *Insert* qui a en paramètre un retour chariot. En effet, cela mettra un retour chariot dans le fichier et compliquera la lecture. Pour contrer ce problème de façon rapide, on remplace les retours chariots en paramètres des commandes *Insert* par un caractère non attribué ("n̂ par exemple).

La commande script fait partie des deux commandes non enregistrées, car son fonctionnement actuel demande un isolement par rapport au système de mémoire.

Pour la lecture des scripts, on implémente cette routine dans la commande *Load*. Lors de son exécution, cette commande va tenter de lire le fichier demandé par l'utilisateur (elle retournera une erreur en cas de fichier inexistant). On crée ensuite un scanner qui va lire le fichier ligne par ligne, pour chaque ligne, on appelle la méthode *summonCommand()*. Cette fonction reçoit une commande sous forme de string, on utilise donc des switch cases pour chaque type de commande, puis on crée un nouvel objet commande adéquate avec les paramètres récupérés, puis on exécute cette commande. À la fin de l'exécution, on sauvegarde l'état final dans l'*Originator*.

3.4 Amélioration de l'interface graphique et débogage

Dans un premier temps, l'interface graphique de la V2 ressemblait à celle de la V1 mais avec un nouveau bouton par fonctionnalité. Ceci n'était pas du tout ergonomique alors des changements ont étés faits. C'est pourquoi l'implémentation de *GUI* dans la V2 n'est plus *MyGUI* mais *MyGUIV2*, cela permet de respecter les consignes de façon stricte (on touche le moins possible à la V1) et cela met en valeur l'utilisation des interfaces qui permettent cette manipulation.

Tout d'abord, les boutons ont tous migrés vers une barre de menu, ce qui rendait l'interface plus agréable visuellement. Pour aller plus loin dans ce sens, des icônes fait mains ont étés ajoutés pour chaque élément du menu. De plus, chaque élément du menu édit à son propre raccourci clavier :

Copier : crtl + F1
Couper : crtl + F2
Coller : crtl + F3
Undo : crtl + F4
Redo : crtl + F5

Les curseurs ont aussi évolués, ils sont maintenant synchrones, ce qui veut dire que si les deux curseurs sont au même endroit, ils le resteront tant que l'utilisateur n'en fait pas explicitement la demande. De plus, le système de *JTextfield* pour les mouvements des curseurs a disparu, maintenant l'utilisateur peut utiliser les touches ← et → pour déplacer le premier curseur ainsi que les touches ↑ et ↓ pour déplacer le deuxième curseur. Lors d'une sélection, un label en bas de l'interface indiquera la sélection en cours, celle-ci sera réinitialiser pour toute commande agissant sur une sélection (*Copy*, *Cut* et *Replace*). À noter que le deuxième curseur ne peut plus être devant le curseur principal maintenant, c'est simplement un choix de design d'imposer une sélection vers l'arrière.

Pour finir, une icône a été ajoutée pour l'application. Celui-ci a servi à se familiariser avec la gestion des paths lors de la compilation et l'exécution avant de faire le système de script (et aussi parce que l'icône à défaut n'est pas très joli). Cette icône n'est pas fait main, mais vient de https://emojiterra.com/shark/.

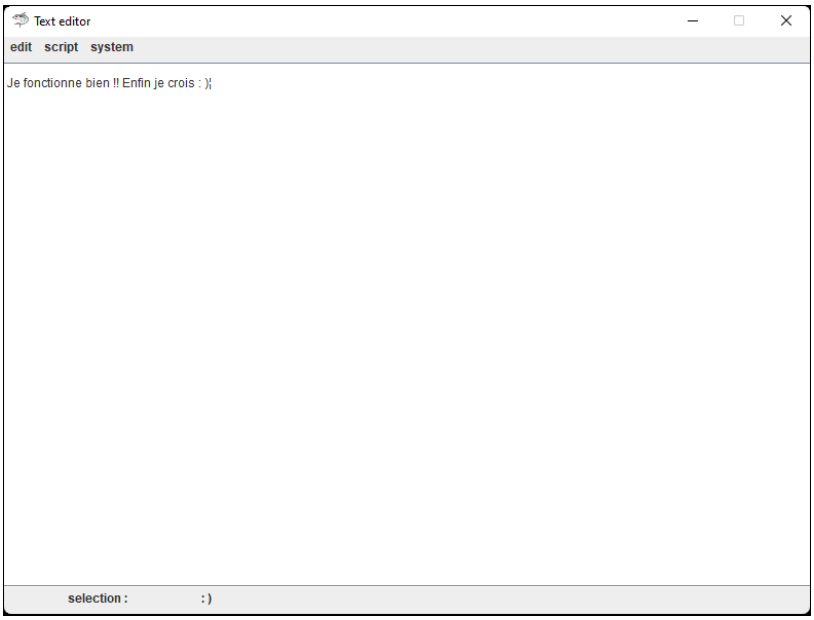

FIGURE 1 – Rendu final de l'application

4 Conclusion

Dans ce rapport, nous avons mis en place l'implémentation de la conception faite précédemment. Cette implémentation est le reflet de la conception et donc est une façon de faire parmi d'autres. L'utilisation des interfaces s'est avérée très utile lors du passage de la V1 à la V2 et donc sera un choix de conception à réutiliser sur d'autres projets à l'avenir (surement avec une factory pour rendre le tout encore plus puissant). Toutefois, il a été compliqué de placer l'Originator dans la V2 en restant au maximum dans les critères SOLID mais cela n'a pas freiné l'amélioration du comportement de l'application pour autant.