Master 1 Informatique

ACO - TP BOUSSAA Mohamed

FEREY Antoine - Groupe 1.2

07 Décembre 2016

Table des matières

Version 1

Cette version comporte les commandes basiques demandées. Le projet comporte deux commandes supplémentaires : *Chargement* et *Sauvegarde.*

Pattern commande

ConcreteCommande

Il y a dans la version 1, 8 commandes au total :

- Sélection
- Insertion de texte
- Suppression
- Copier
- Couper
- Coller
- Chargement
- Sauvegarde

Toutes les commandes vont appeler une méthode du Receiver :
Exemple :

```
@Override
publicvoidexecute() {
  this.editeur.couper() ; // execute le traitement de la commande couper de l'editeur
}
```

L'Invoker - Client

Dans le projet, la classe Client et la classe Invoker du pattern Command sont représenté en une seul classe : IHM. Cette avec cette dernière que l'utilisateur va interagir. Cette classe contient donc le Receiver et toutes les ConcreteCommandes. Il possède les méthodes pour exécuter celles-ci.
Exemple :

```
@Override
voidpressCopier(){
this.copier.execute() ; // execute la commande copier
}
```

Receiver

Le Receiver du pattern Command est modélisé par l'éditeur. C'est là que tous les traitements des commandes se font.
Il contient différents objets pour les traitements :

- Presse papier → Le presse papier pour les commandes copier,couper,coller.
- Buffer → contient le texte de l'éditeur
- Selection → représente le curseur
- FluxFile → Gère les manipulations des fichiers lors de la sauvegarde et du chargement.

Les Comportements des commandes dans l'éditeur :

Sélection

La sélection peut représenter un curseur ou la sélection d'une portion de texte.

Insertion de texte

Insère un caractère dans le buffer après le curseur. Si il y a une sélection du texte, il supprime la sélection et insère le caractère à la place. La sélection est avancé après le caractère dans les deux cas.

Suppression

Supprimer le caractère derrière le curseur. La sélection est décrémenté de un.

Si il y a une sélection du texte il supprime le texte. La sélection se situe au curseur de début dans ce dernier cas.

Copier

copie la sélection dans le presse papier. Si la sélection est un curseur, il ne se passe rien.

Couper

copie la sélection dans le presse Papier puis supprimer la sélection. Si la sélection est un curseur, il ne se passe rien.

Coller

Il ne se passe rien si le presse papier est vide.

Insère le presse papier dans le buffer après le curseur. Si il y a une sélection du texte, il le supprime et insère le presse papier à la place. La sélection se situe après le texte ajouté dans les deux cas.

Chargement

charge un fichier depuis le dossier sauvegarde dans le buffer

Sauvegarde

sauvegarde le buffer dans un fichier situé dans le dossier sauvegarde

La classe Pressepapier

Le classe PressePapier contient juste un string qui change en fonction des actions copier ou couper.

La classe Selection

Comporte deux int qui représentent le curseur de début et de fin. C'est dans cette classe que l'on gérer si les curseurs ne sont pas cohérent.

Cas Incohérent --> Solution

1. le début est supérieur à fin --> on inverse les deux valeurs
2. fin est inférieur à début --> on inverse les deux valeurs
3. fin est supérieur à la taille du buffer --> fin est ramené à la fin du buffer
4. début est supérieur à la taille du buffer --> début est ramené à la fin du buffer
5. début est inférieur à 0 --> début est ramené à 0
6. fin est inférieur à 0 --> fin est ramené à 0

La classe FluxFile

Classe qui Gère le chargement et la sauvegarde d'un fichier texte.

La classe Buffer

Cette classe contient le contenue texte de l'éditeur. Mon choix de variable s'est orienté sur type StringBuffer. Contrairement au type String qui est un objet non mutable, l'objet de type StringBuffer est mutable (sa valeur peut changer au cours du temps).

Buffer gère les modifications du texte dans le StringBuffer. C'est à dire, qu'elle vérifie que l'insertion de caractères ou de String, ou encore la suppression dans le StringBuffer se passe bien. Si cela se passe mal, la classe lève une erreur personnalisé : **ErreurInsertionException**. Cette erreur est alors attrapée et gérer par la classe de l'éditeur.

Version 2

Dans cette version 2, on ajoute la fonctionnalité d'enregistrer des macros. Nous avons donc trois commandes qui s'ajoutent aux huit premières :

- Start : Commence l'enregistrement de la macro. Elle efface la macro précédemment enregistré si il y en avait une.
- Stop : Stop l'enregistrement de la macro.
- Rejouer :Rejoue la macro enregistrer si il y en a une.

Toutes les commandes ne s'enregistrent pas pour les macros. Les commandes ci-dessus par exemple ne seront pas ajoutées dans l'enregistreur. Enregistrer des enregistrements de macro n'est pas très banal et agréable pour l'utilisateur. Les commandes sauvegarder et charger ne sont pas non plus enregistrable.

Pattern memento

Originator

Il s'agit des commandes à enregistrer pour pouvoir les rejouer. Chaque Commande va retourner un memento lors de l'enregistrement (`getMemento()`). Quand l'action *rejouer* sera enclenchée, la commande reprendra son memento(`setMemento(ItfMemento m )`) et sera de nouveau exécutée.

Memento

Les mementos ne sont pas toujours obligatoire pour pouvoir rejouer une commande. Par exemple la commande Suppression s'exécute sans avoir besoin de savoir quelque chose vu que la sélection est un traitement à pars.

Il y a seulement deux commandes qui ont besoin de savoir quelque chose pour s'exécuter correctement :

- InsertionTexte → Quelle caractère à insérer ? → un memento avec 1 caractère.
- Selection → Quelle est la position du curseur ? → un memento avec le curseur de début et le curseur de fin.

Pour répondre à ce problème, un memento est enregistrer avec les données qu'il faut pour que la commande s'exécute bien.

Pour toutes les autres commandes, un memento vide est enregistrer.

Caretaker

Le Caretaker correspond à l'enregistreur dans le projet. L'enregistreur possède comme attribut un boolean pour savoir s'il doit enregistrer les actions de l'utilisateur et deux listes très liées :

- Une liste de commande
- Une liste de memento

Quand on enregistre une commande, on l'ajoute à la première liste et on enregistre son memento dans la deuxième :

```
publicvoidAjout(ItfCommande commande) {
  if(this.rec) {
    this.listcommande.add(commande);
    this.listmemento.add(commande.getMemento());
  }
}
```

Il se passe la même chose lorsqu'on rejoue une macro. On parcours les deux listes simultanément pour rejouer les commandes avec leurs mementos.

Version 3

Dans cette version 3, on ajoute la fonctionnalité de faire et défaire les actions. Nous avons donc deux commandes qui s'ajoutent aux anciennes :

- Undo : annule la dernière action. Si répété plusieurs fois, on peut revenir au début
- Do : refaire la dernière action annulé.

Les commandes Do et Undo ne peuvent annuler ou refaire certaine action.

Tout les commandes en rapport avec l'enregistrement ne peuvent être refaire ou défaire.

Le contraire est aussi vrai, l'enregistreur n'enregistre pas les commandes Do et Undo.

Pattern memento

Originator

J'ai choisie d'enregistrer l'état actuel de l'éditeur à chaque changement du buffer. Cela représente le choix de sacrifier le stockage pour gagner du temps de traitement. L'originator de cette partie est donc l'éditeur.

Memento

Il n'y a donc une seule classe memento pour cette partie. Le memento MementoEtatBuffercomporte l'état du buffer et la sélection.

Caretaker

Le Caretaker de la partie 3 se nomme TimeMachine. Nom approprié pour retourner dans le passé et revenir dans le présent. La TimeMachine comporte deux attributs majeurs.

- Une pile des états passés
- Une pile des états futurs

A chaque commande Undo, la tête de la pile avant se déplace sur la tête de la pile après.

A chaque commande Do, la tête de la pile après se déplace sur la tête de la pile avant.

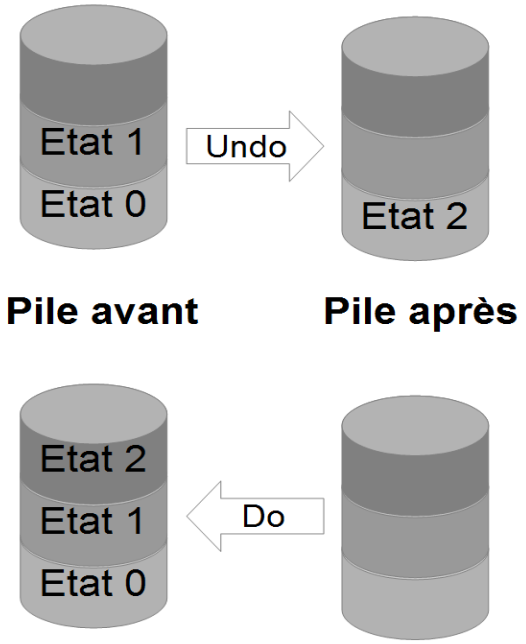

Schéma des deux piles dans la TimeMachine

Pattern Observer

Dans la partie Originator nous avons dit : *« enregistrer l'état actuel de l'éditeur à chaque changement du buffer »* . Pour savoir quand le buffer est modifié, le pattern Observer me paraît tout à fait approprier pour résoudre ce problème. J'utilise les classes Observer et Observable déjà implémentées par Java pour réaliser le pattern.

Subject

L'observable est l'éditeur, celui qui fait les traitements, donc il sera quand le buffer sera modifé. On notifie les observers quand il y a :

- Insertion d'un caractère
- Suppression
- Couper quand il y a une sélection
- Coller quand le presse-papier n'est pas vide
- Chargement d'un fichier

Observer

Le Caretaker (TimeMachine) est notifié lors d'un changement du buffer. Nous ajoutons alors le memento à la pile avant.