ECOLE SUPERIEURE D'INGENIEUR DE RENNES

Université de Rennes

TP2 - Mini éditeur de texte

Pierre BERTHOLOM
Florian CLIQUET

OMD S7

3 novembre 2025

Remarque générale

Nous attestons que le contenu de ce document est original et est issu de nos réflexions personnelles

Sommaire

Introduction

Dans ce TP, nous avons pour projet de réaliser un mini-éditeur de texte en C++ en suivant une démarche de conception orientée-objet. Le cahier des charges impose une conception en deux versions :

- **Version 1 :** Implémentation des fonctionnalités de base d'édition (buffer, sélection, copier, couper, coller).
- **Version 2 :** Ajout des fonctionnalités avancées d'enregistrement/rejeu (macros) et de défaire/refaire (undo/redo).

Nous avons choisi de développer un éditeur de texte en mode terminal (TUI) à la manière de Vim ou Nano. Pour ce faire, nous utilisons la bibliothèque C `ncurses`, pour la gestion de l'interface, tout en encapsulant sa logique dans une classe C++ dédiée (`NcursesView`).

Ce rapport se concentre sur la conception et le développement des deux versions successives du logiciel.

1 Dossier de Conception (V1)

1.1 Architecture Globale

Nous avons structuré l'application selon une architecture dérivée du patron de conception **MVC** (Modèle-Vue-Contrôleur) :

- **Le Modèle :** Représente les données et la logique métier de l'éditeur. Il est constitué de l'ensemble de classes `PieceTable`, `Buffer`, `Selection` et `Clipboard`. Il ne connaît pas la Vue.
- **La Vue :** Gérée par la classe `NcursesView`. Elle est responsable de l'affichage de l'état du Modèle dans le terminal et de la capture des entrées utilisateur.
- **Le Contrôleur :** Le rôle de contrôleur est partagé. La classe `NcursesView` capture une entrée (ex : `CTRL+C`) et la traduit en un appel de méthode (ex : `editor.copy()`). La classe `Editor` agit comme un contrôleur principal ou une "façade" pour le Modèle, coordonnant les actions entre la `PieceTable` et la `Selection`.

Grâce à ce modèle, la logique de l'éditeur (`Editor`, `PieceTable`) est totalement indépendante de la technologie d'affichage (`ncurses`).

1.2 Choix de conception : Buffer

1.2.1 Conception Initiale (Buffer Simple)

Notre première conception (Figure 1) reposait sur une classe `Buffer` contenant une unique `std::string`.

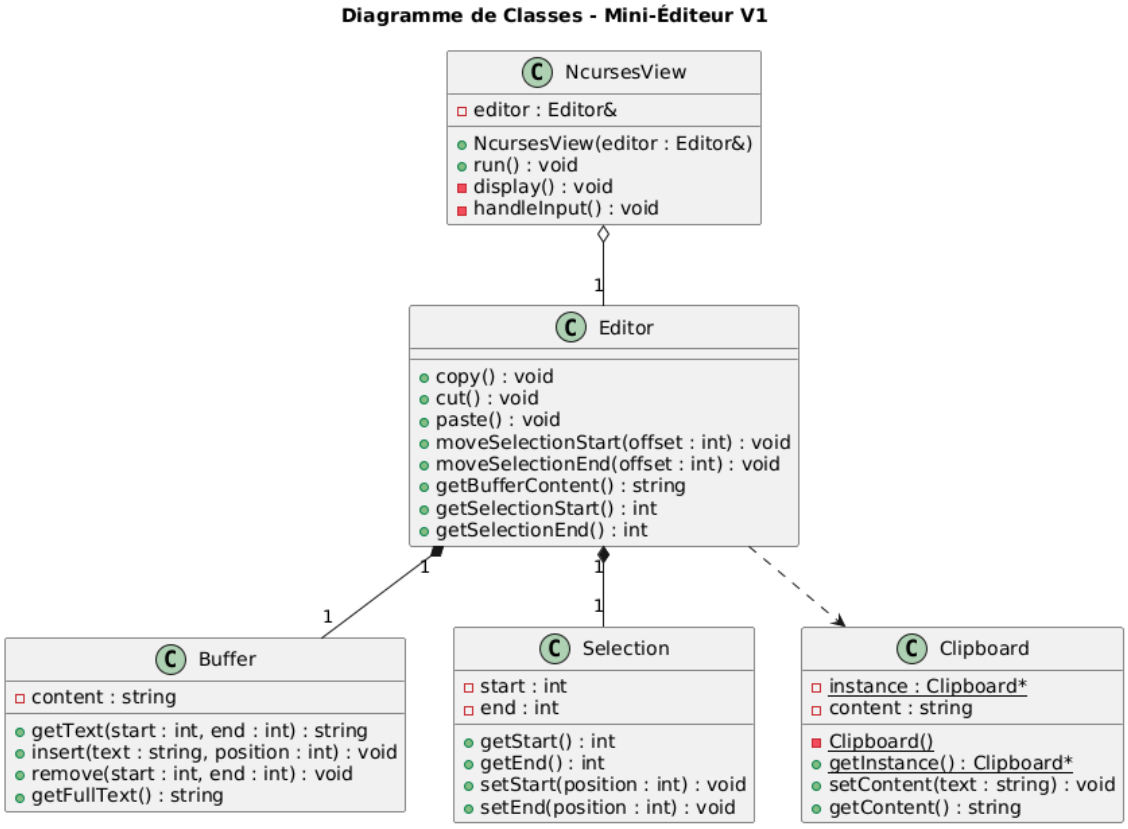

FIGURE 1 – Diagramme de classes initial (non retenu)

Problématique : Cette approche est simple mais très inefficace. Une insertion ou suppression au milieu d'un fichier très volumineux demanderait la copie de millions de caractères en mémoire, rendant l'éditeur inutilisable.

Nous avons exploré plusieurs alternatives pour le stockage du texte :

— **Gap Buffer :** Cette structure maintient le texte en deux morceaux séparés par le curseur, formant un trou (gap). C'est plus performant qu'une `std::string` pour les éditions localisées. Cependant, comme pour un buffer simple, il ne facilite pas l'implémentation de l'undo/redo ou des macros.

— **Rope (Corde) :** Une structure de données en arbre binaire où les feuilles contiennent des sous-chaînes de caractères. Elle est extrêment performante

pour les opérations sur des fichiers volumineux. Mais nous avons jugé cette structure "overkill" et d'une complexité d'implémentation trop élevée pour les objectifs du projet.

— **Piece Table (Solution Retenue) :** Cette structure utilise deux buffers : un buffer immuable contenant le texte original du fichier (`originalBuffer`) et un buffer contenant toutes les insertions successives (`addBuffer`). Le document actuel est représenté par une liste de "morceaux" (`Piece`) qui pointent simplement vers des segments de l'un de ces deux buffers.

Nous avons retenu la **Piece Table** car elle représente un excellent compromis :

1. **Performance :** Les insertions et suppressions sont quasi-instantanées. Elles ne modifient que la liste des `Piece`, sans jamais copier le texte existant.
2. **Facilité pour la V2 :** L'implémentation de l'undo/redo est grandement facilitée. Le buffer original étant immuable, les données ne sont jamais perdues.

1.3 Diagramme de Cas d'Utilisation (V1)

Ce diagramme présente les fonctionnalités de base de la V1 du point de vue de l'acteur Utilisateur.

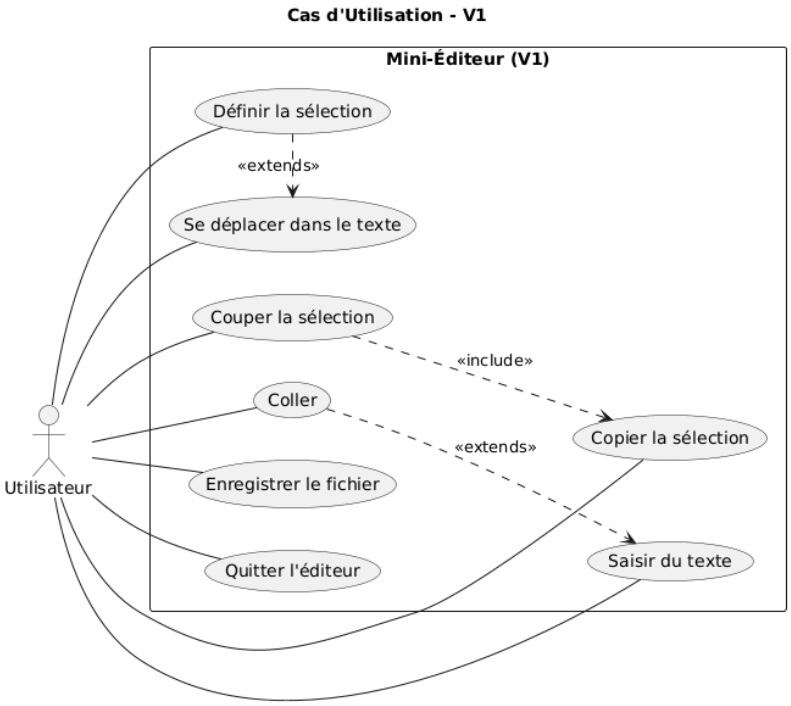

FIGURE 2 – Diagramme de Cas d'Utilisation (V1)

1.4 Diagramme d'États (V1)

Ce diagramme modélise les états de l'interface utilisateur. En V1, l'état principal est binaire : soit l'utilisateur déplace un simple curseur (état `Édition_Curseur`), soit il étend activement une sélection (état `Édition_Sélection`).

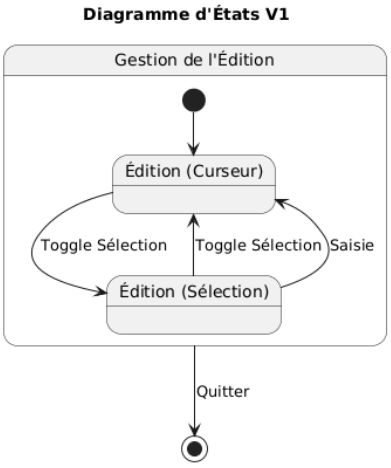

FIGURE 3 – Diagramme d'États (V1)

1.5 Diagramme de Séquence (V1)

Le diagramme de séquence suivant décrit l'insertion de texte. L'appel est direct, `NcursesView` notifie l'`Editor`, qui appelle directement la méthode `insert()` de la `PieceTable`. La suppression de texte agit de façon similaire.

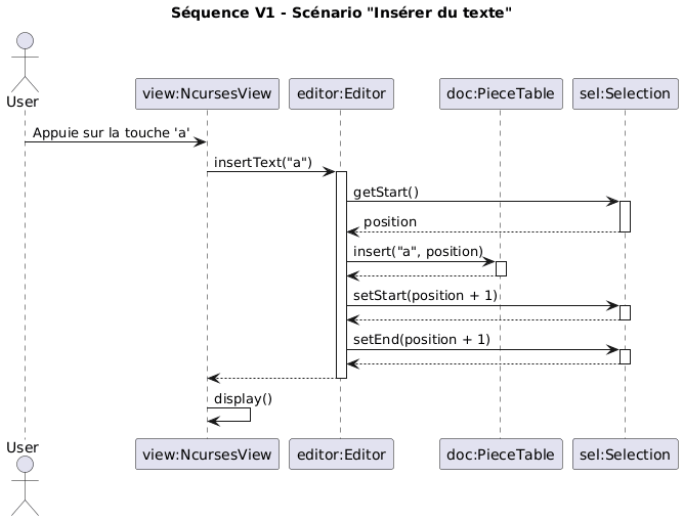

FIGURE 4 – Diagramme de Séquence (V1) - Insertion de texte

1.6 Diagramme de Classes Final (V1)

L'adoption de la Piece Table conduit au diagramme de classes final pour notre Version 1. Il montre la séparation claire entre la Vue (`NcursesView`) et le Modèle (`Editor`, `PieceTable`, `Selection`).

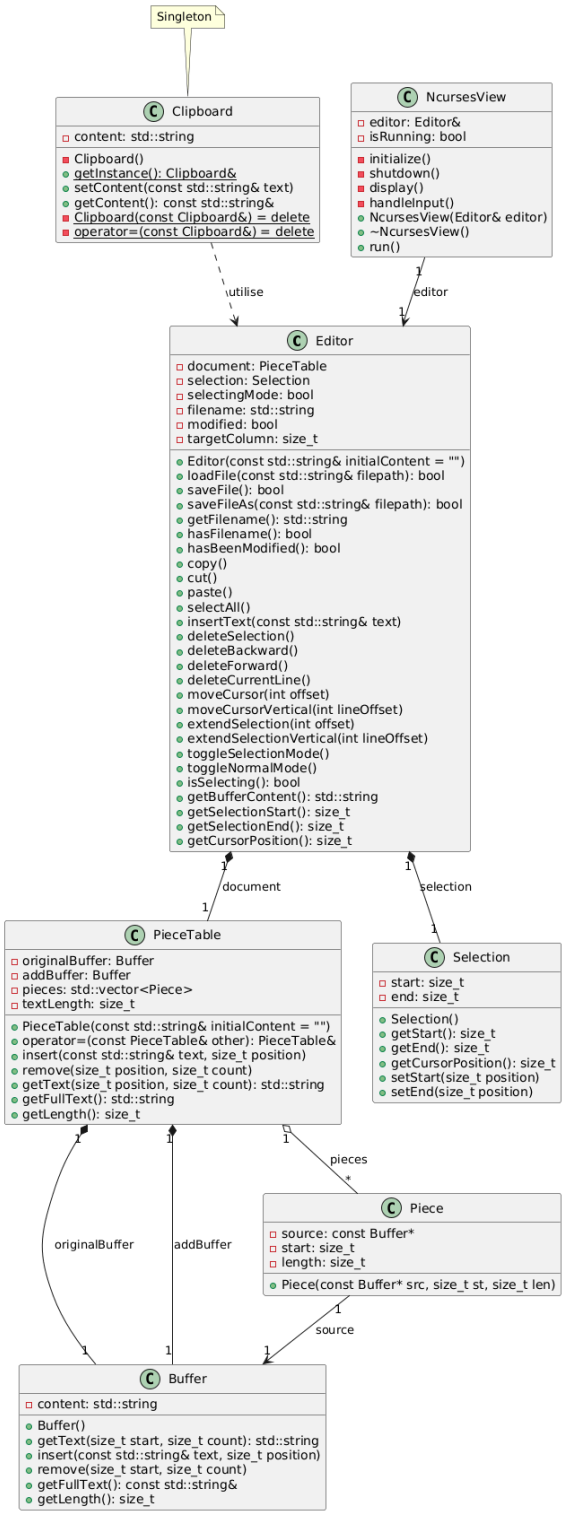

FIGURE 5 – Diagramme de classes final de la V1

1.6.1 Description des Classes Principales

Editor : C'est la classe centrale du logiciel, la façade de notre Modèle. Elle manipule les interactions entre la `PieceTable` (le contenu) et la `Selection` (le curseur). Elle implémente les méthodes telles que `copy()`, `insertText()` que la Vue peut appeler sans connaître les détails d'implémentation et l'accès de la `PieceTable`.

PieceTable : Coeur de la logique de stockage. Elle contient les deux `Buffers` et la liste des `Piece`. C'est elle qui implémente la logique d'insertion et de suppression en manipulant cette liste.

Buffer : Classse utilitaire pour agir comme conteneur de `std::string`. Elle est utilisée pour les deux buffers de `PieceTable` pour stocker le texte original et le texte ajouté.

Selection : Représente la sélection de texte. Elle stocke simplement un index de début et de fin pour la sélection. Un curseur simple correspond à une sélection où `start == end`.

Clipboard : Implémenté en tant que **Singleton**. Cela garantit qu'une seule et unique instance de presse-papiers existe dans l'application.

NcursesView : La couche d'abstraction pour l'interface. Elle initialise et ferme `ncurses`, gère la boucle d'événements principale (via `getch()` dans `handleInput()`), affiche le contenu de l'éditeur à l'écran, et traduit les entrées clavier en appels de méthodes sur l'instance d'`Editor`.

2 Dossier Développeur (V1)

2.1 Structure du Projet

Le projet est organisé de manière standard en C++, en séparant les fichiers d'en-tête (interfaces) des fichiers sources (implémentations).

```
mini-editeur/
|
+-- include/                # Fichiers d'en-tête (.hpp)
|   |-- Buffer.hpp
|   |-- Clipboard.hpp
|   |-- Editor.hpp
|   |-- NcursesView.hpp
|   |-- Piece.hpp
|   |-- PieceTable.hpp
|   `-- Selection.hpp
|
```

```
+-- src/                    # Fichiers sources (.cpp)
|   |-- Buffer.cpp
|   |-- Clipboard.cpp
|   |-- Editor.cpp
|   |-- main.cpp            # Point d'entrée
|   |-- NcursesView.cpp
|   |-- PieceTable.cpp
|   `-- Selection.cpp
|
+-- CMakeLists.txt
`-- README.md
```

2.2 Compilation

Le projet est compilé en C++14 en utilisant CMake, via un `CMakeLists.txt`. Le projet contient une unique dépendance externe, la bibliothèque `ncurses`.

3 Dossier de Conception (V2)

Dans la version 2, nous allons implémenter la fonctionnalité de macro et la fonctionnalité de défaire/refaire (undo/redo). Notre conception basé sur une `PieceTable` et notre architecture de la V1, fournissent une base solide pour cette extension.

3.1 Patron de Conception Commande (Command Pattern)

Pour implémenter l'undo/redo et les macros, nous avons utilisé le design de **Command pattern**. L'idée est d'encapsulé les actions de l'utilisateur dans des commandes (objets).

- **Pour l'Undo/Redo :** Nous avons défini une interface `Command` avec les méthodes `execute()`, `undo()` et `clone()`. Les actions qui modifient le document (comme `InsertCommand` et `DeleteCommand`) implémentent cette interface.
- Un `CommandManager` centralise la logique d'historique. Il possède une `undoStack` et une `redoStack`, toutes deux des piles de `std::unique_ptr<Command>`.
- **Undo :** Dépiler de `undoStack`, appeler `undo()` sur la commande, et la transférer sur `redoStack`.
- **Redo :** Dépiler de `redoStack`, appeler `execute()` sur la commande, et la transférer sur `undoStack`.
- **Pour la macro :** Nous avons adopté une approche hybride. Plutôt que de stocker les objets `Command` (qui ne couvrent pas les actions sans "undo", comme le mouvement du curseur), le `CommandManager` stocke un `std::vector<std::function<void()»`.
- Lors de l'enregistrement, chaque action de l'utilisateur (y compris les mouvements) est encapsulée dans une lambda C++ et stockée dans ce vecteur via `recordAction()`. Rejouer la macro consiste simplement à itérer sur le vecteur et à invoquer chaque `std::function`.

3.2 Diagramme de Cas d'Utilisation (V2)

Pour la V2, le diagramme est étendu pour inclure les nouvelles fonctionnalités avancées : le défaire/refaire (Undo/Redo) et l'enregistrement/rejeu de macros, comme requis par le sujet.

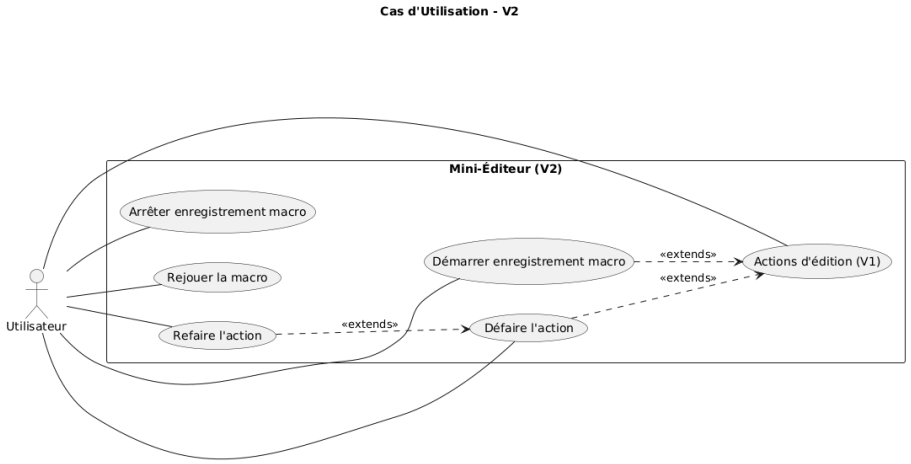

FIGURE 6 – Diagramme de Cas d'Utilisation (V2)

3.3 Diagramme d'États (V2)

Le diagramme d'états de la V2 devient plus complexe. Nous utilisons des états parallèles pour modéliser le fait que l'enregistrement d'une macro est un état indépendant de l'état d'édition (curseur ou sélection).

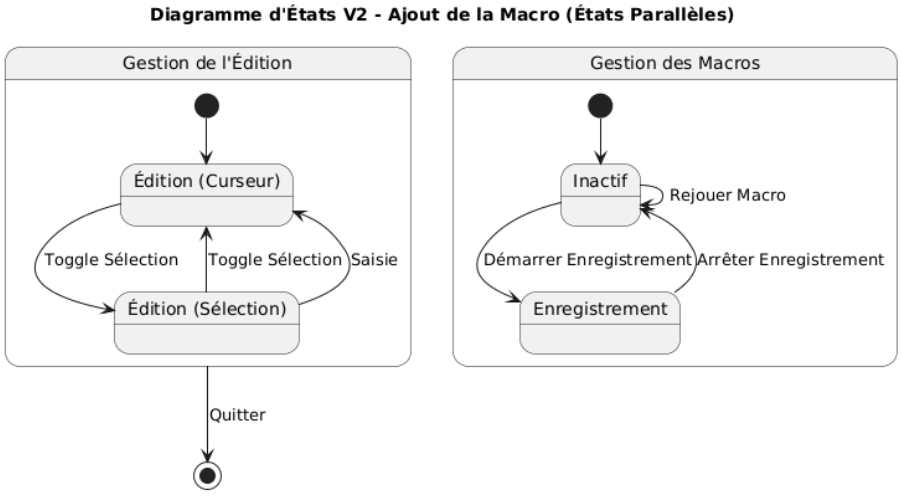

FIGURE 7 – Diagramme d'États (V2) - Ajout des macros (États parallèles)

3.4 Diagramme de Séquence (V2)

Avec le Command Pattern, le même scénario d'insertion de texte est modifié. L'`Editor` ne fait plus le travail lui-même, il crée un objet Commande (`InsertCommand`) et le délègue au `CommandManager`, qui orchestre l'exécution et le stockage pour l'historique. La suppression agit de manière similaire avec la création d'un objet `DeleteCommand`.

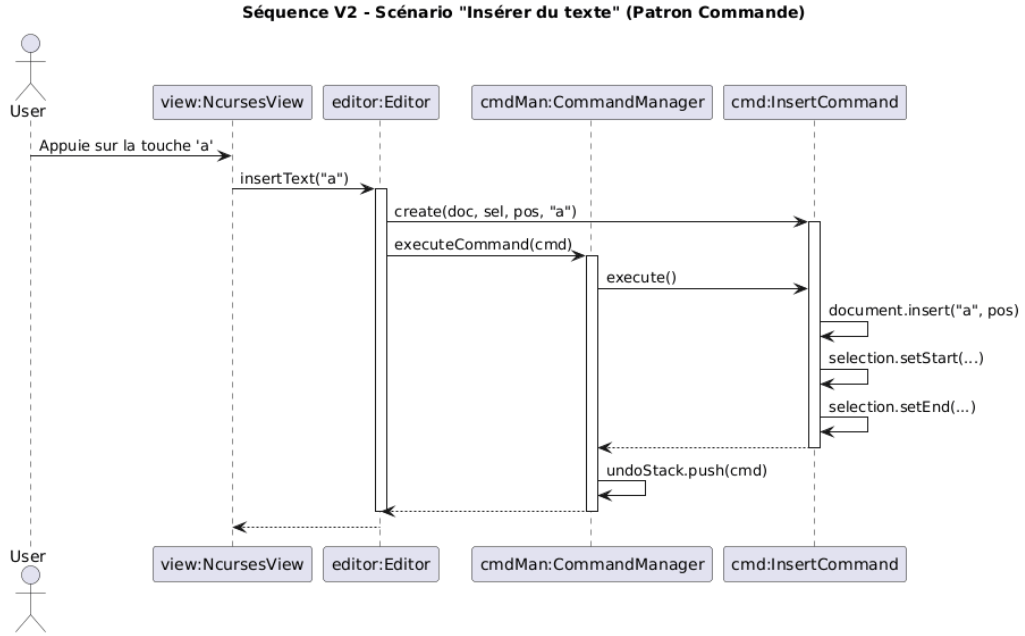

FIGURE 8 – Diagramme de Séquence (V2) - Insertion de texte

3.5 Diagramme de Classes (V2) - Final

Ce diagramme intègre le `CommandManager` et les classes du patron Commande (l'interface `Command` et ses implémentations `InsertCommand` et `DeleteCommand`) dans l'architecture de la V1.

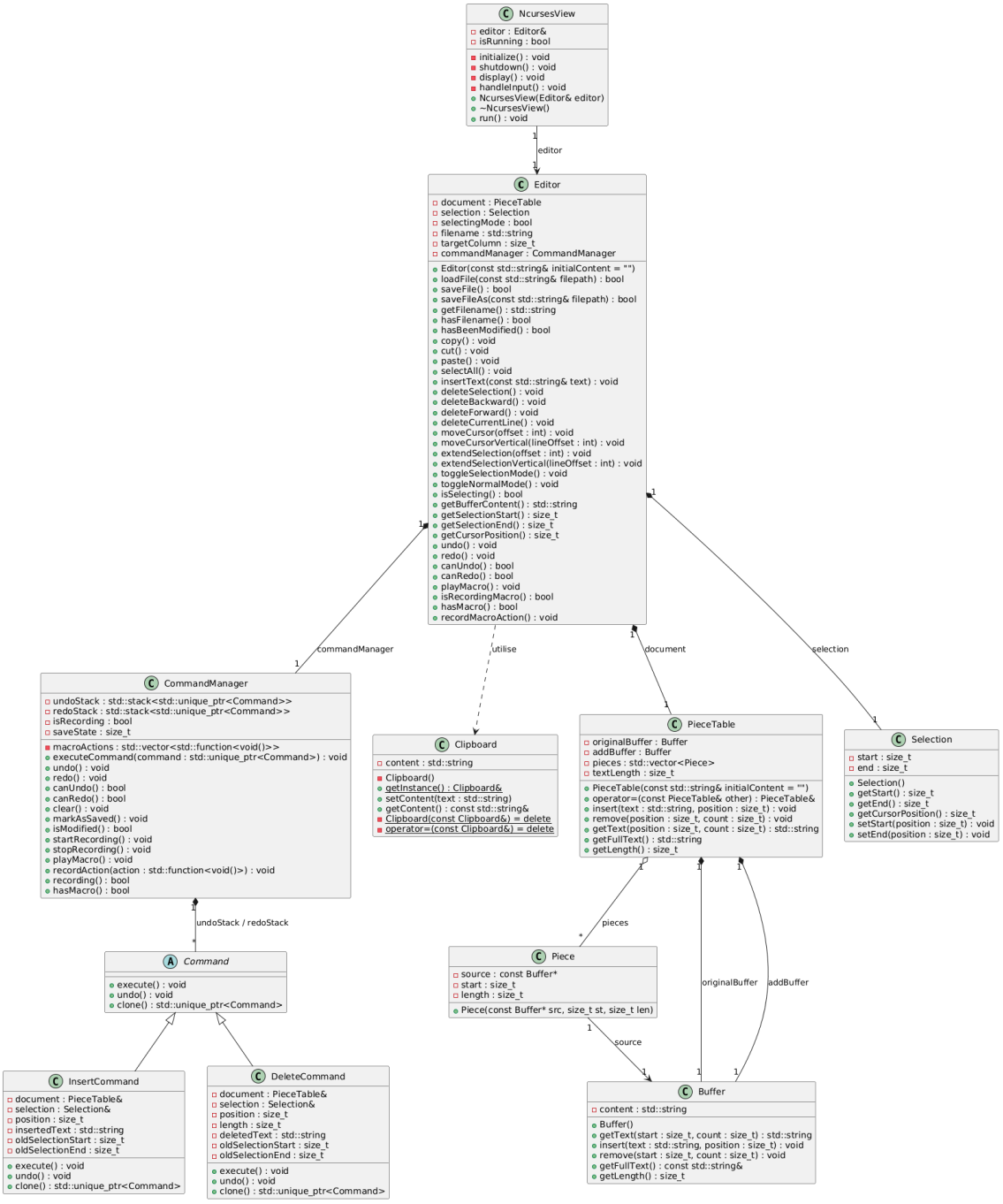

FIGURE 9 – Diagramme de classes final de la V2 intégrant le Patron Commande

4 Dossier Développeur (V2)

L'implémentation de la V2 a nécessité certaines modifications de la classe `Editor` et l'ajout d'un `CommandManager` pour gérer la logique d'exécution des actions.

4.1 Implémentation des Commandes

Le changement principal est que la classe `Editor` ne modifie plus directement la `PieceTable` ou la `Selection`. Son rôle est désormais de **créer un objet Commande** et de le déléguer au `CommandManager`.

Toutes les méthodes d'édition (comme `insertText`, `deleteSelection`, `paste`) suivent ce nouveau schéma. Elles instancient une commande concrète (`InsertCommand` ou `DeleteCommand`) et appellent `commandManager.executeCommand()`.

4.2 Gestion de l'Historique (Undo/Redo)

Le `CommandManager` gère l'historique. L'utilisation de `std::unique_ptr` garantit que chaque objet Commande a un propriétaire unique et que sa mémoire est gérée automatiquement.

Lorsque `executeCommand` est appelée, la propriété de la commande est transférée (via `std::move`) dans l'`undoStack`. Les appels à `undo()` et `redo()` transfèrent simplement la propriété de ce pointeur entre les deux piles.

Nous avons également implémenté la gestion de l'état "modifié" (`isModified()`) en utilisant un marqueur (`saveState`) qui mémorise la taille de l'`undoStack` lors de la dernière sauvegarde, permettant une vérification instantanée.

4.3 Implémentation des Macros (Enregistrement/Rejeu)

Pour les macros, une solution plus flexible était nécessaire car nous voulions enregistrer toutes les actions, y compris les mouvements de curseur qui ne sont pas des objets `Command` (car ils n'ont pas d'état `undo`).

Nous avons utilisé des `std::function` et des expressions lambda.

1. La classe (`NcursesView` ou `Editor`) qui gère l'entrée clavier appelle `commandManager.recordAction()` **avant** d'exécuter l'action.
2. Elle passe une lambda qui capture le contexte nécessaire (`[this]`) et encapsule l'appel de méthode exact.
3. Si `isRecording` est vrai, le `CommandManager` stocke cette lambda dans un `std::vector`.
4. `playMacro()` itère simplement sur le vecteur et exécute chaque lambda stockée.

Cette approche hybride permet un système robuste pour l'undo/redo et un système d'enregistrement flexible pour les macros.

5 Conclusion

Nous avons pu à travers ce TP, concevoir et implémenter un éditeur de texte performant. La conception basée sur une séparation Modèle-Vue et l'utilisation d'une `PieceTable`, s'est avérée robuste et performante. Elle a fourni une base solide qui a permis l'implémentation des fonctionnalités de la Version 2.

L'intégration du Command Pattern, bien que complexe, a rempli tous les objectifs du projet, en fournissant un système d'historique (undo/redo) et de macros à la fois performant, extensible et conforme aux exigences de conception orientée-objet.