Université de Nantes
Faculté des sciences et Techniques
Master 1 ALMA

Mini-éditeur de texte
Rapport du projet Génie Logiciel

Yvan NOBLET
Julien OUVRARD

Promoteur :
Prof. Gerson Sunyé

Décembre 2014

Table des matières

1 Présentation du projet mini-éditeur

C'est dans le cadre de l'UE de Génie Logiciel, que nous avons été amenés à travailler sur le projet d'un mini-éditeur de texte. Le projet portait sur le point de vue structurel du logiciel, c'est pourquoi il ne sera pas question d'interface graphique utilisateur mais de fonctionnalités et de conception. Ce travail, à réaliser en binôme, nous incite à coder proprement via des schémas reconnus et de comprendre puis tracer le chemin de la spécification à l'implémentation en passant par la conception.

Le mini-éditeur de texte propose toutes les commandes de base d'un éditeur classique telles que les effets copy, cut, paste, les effets undo/redo et bien évidemment write et backspace. Notre éditeur comporte également la fonctionnalité de créer des macros à partir des commandes citées précédemment. De plus, une classe utilisateur a été instanciée. Cette dernière permet de simuler l'accès et la modification d'un buffer par plusieurs utilisateurs qui le partage.

Ainsi, à travers ce rapport nous allons aborder et, si besoin, expliquer le choix des différents logiciels, langages, frameworks et autres patterns utilisés tout au long du projet.

2 Langage & UML

2.1 Scala

Le langage utilisé pour réaliser cet éditeur est le Scala dans sa version 2.11.4. Scala est un langage de programmation multi-paradigme, son nom vient de l'anglais *Scalable language* qui peut être traduit par "langage adaptable". Dans notre cas il est utilisé en tant que langage de programmation orienté objet.
Nous disposons de deux dossiers contenant les sources : src/main et src/test. Dans le premier nous retrouvons le code de chacune des commandes ainsi que celui de l'utilisateur et du buffer, tandis que dans le second se trouve seulement le code concernant les tests.

2.2 Papyrus

Afin de modéliser correctement nos classes en UML, nous avons utilsé l'environnement Papyrus intégré à l'IDE Eclipse.
Papyrus vise à fournir un environnement intégré facile à utiliser pour éditer les modèles de type EMF (Eclipse Modeling Framework), il soutient en particulier UML et les langages de modélisation connexes tels que SysML et MARTE. Papyrus offre également un support très avancé pour les profils UML qui permet aux utilisateurs de définir des éditeurs pour les DSL (Domain Specific Language) basés sur le standard UML 2.

2.3 Tests

Les tests ont été réalisés en suivant le framework JUnit qui prend en charge les tests Scala. Les tests unitaires réalisés portent sur les méthodes des **Commandes**.

3 Fonctionnalités de l'éditeur

3.1 Le Buffer

Buffer

Le *Buffer* représente le cœur de l'éditeur. C'est l'espace de travail avec lequel les différentes commandes vont interagir. Le buffer est constitué d'un *contenu* et d'un *index* et des objets **Clipboard**, **Selectionner** et **Historique** qui seront détaillés par la suite.

Le *contenu* est de type **String** et représente ce qui est écrit dans par les utilisateurs dans le mini-éditeur de texte.

L'*index* est de type **int** et nous sert à nous repérer dans l'historique. Á chaque modification du *contenu* l'index est incrémenté.

Clipboard

Le *Clipboard* représente le presse-papier d'un buffer. C'est ici que sera enregistrée une sélection copiée ou coupée. Le presse-papier reste le même du moment qu'il n'y a pas de nouvelle sélection (copiée ou coupée) qui l'écrase.

Le presse-papier est simplement constitué d'un *contenu* de type **String** qui est une sous-chaîne du contenu du buffer.

Selectionner

L'objet *Selectionner* permet la sélection d'une sous-chaîne du contenu du buffer. Plus précisément, dans le projet c'est un objet constitué de deux index représentant un intervalle et d'une chaîne de caractère représentant la sélection. Si l'intervalle est nul, il s'agit d'un curseur et la chaîne est vide.

Historique

L'*Historique* représente la mémoire du buffer. Il contient toutes les modifications auxquelles le buffer a été sujet. Il permet, grâce aux commandes **Annuler** et **Refaire** (que nous verrons ensuite), de revenir à un état antérieur du buffer via un **Memento**.

Memento

Le *Memento* est l'état d'un buffer entre deux modifications. Il mémorise le contenu (**String**) du buffer ainsi que la position (**int**) de son curseur à cet instant.

3.2 L'Utilisateur

Utilisateur

L'*Utilisateur* est un **Observateur**, c'est-à-dire qu'il est abonné à un ou plusieurs **Buffer** et qu'il sera notifié dès que ceux-ci seront modifiés par un autre utilisateur.

Un utilisateur possède simplement un *nom* de type **String**.

Observateur

L'*Observateur* est une classe abstraite. Un observateur est notifié par l'objet observable auquel il est abonné lorsque ce dernier le souhaite.

Observable

L'*Observable* est une classe abstraite. Un observable contient une liste d'**Observateurs** qui y sont abonnés.

3.3 Les commandes

Commande

La classe abstraite *Commande* permet une généricité des exécutions des commandes qui suivent, en particulier lors de l'utilisation de **Macros**.

Ecrire

L'objet *Ecrire* permet simplement d'ajouter du contenu de type **String** dans le buffer suivant la position du curseur.

Effacer

L'objet *Effacer* efface soit un caractère du contenu du buffer, soit une sélection si une sélection a été faite au préalable.

Copier

L'objet *Copier* sauvegarde la sélection (de type **String**) dans le **Clipboard**.

Couper

Comme l'objet **Copier**, *Couper* sauvegarde la sélection dans le **Clipboard**, en revanche il effectue aussi une suppression de la sélection dans le contenu du buffer.

Coller

Coller permet d'ajouter le contenu du **Clipboard** au contenu du **Buffer** selon la position du curseur ou à la place d'une sélection.

Annuler

La commande *Annuler* propose la possibilité à l'utilisateur de revenir en arrière dans ses modifications du buffer. Grâce à l'**Historique** le buffer peut reprendre un état passé.

Refaire

La commande *Refaire* permet le déplacement inverse de la commande **Annuler** et ainsi revenir à l'état présent du buffer, toujours grâce à l'**Historique**.

Macro

Finalement la *Macro* offre la possibilité à l'utilisateur de combiner plusieurs des commandes vues précédemment en une seule nouvelle commande.

Cet objet contient une liste de **Commande** qui s'exécuteront les unes à la suite des autres en un seul appel selon la configuration choisie.

4 Patrons de conceptions

4.1 Memento

Le patron de conception Memento nous a été utile pour implémenter les commandes **Annuler**(undo) et **Refaire**(redo). Il se compose des classes **Historique**, **Memento** et **Buffer**. La classe *Memento* sauvegarde l'état du *Buffer* et la position du curseur. La classe *Historique*, quant à elle, sauvegarde une liste de Memento et permet ainsi d'aller et venir dans les différents états sauvegardés. Enfin, la classe *Buffer*, qui représente l'espace de travail, est la classe dont on enregistre l'état courant à chaque modification (toute modification affectant son contenu). Le pattern permet initialement de récupérer n'importe quel état par son index. Dans ce projet il est utilisé de manière bien spécifique puisqu'on ne se déplace que par états suivants ou précédents pour respecter les fonctionnalités des commandes *Annuler* et *Refaire*.

Une fois implémentées, les commandes permettent d'annuler ou de refaire une(des) opération(s) à volonté.

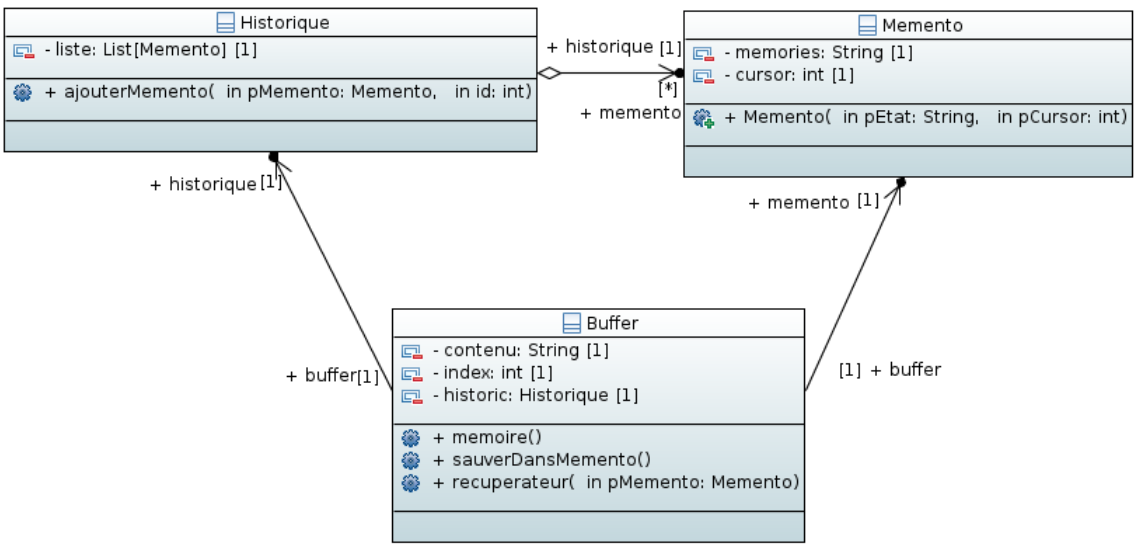

FIGURE 1 – Diagramme UML du patron de conception *Memento*

Malgré un ajustement énoncé plus haut, l'utilisation et l'implémentation de ce patron de conception fut assez évidente avec le principe d'historique. Il garde en effet tout en mémoire, ce qui est ce que nous recherchons lorsque nous souhaitons pouvoir nous déplacer à travers les différents états qu'a pu avoir un objet.

4.2 Observer

Le patron Observer permet à plusieurs **Utilisateurs** abonnés à un même **Buffer** d'être notifiés à chaque mise à jour du contenu du buffer, de ce fait lorsqu'un utilisateur écrira dans un buffer, tous les utilisateurs abonnés à ce buffer le verront. De plus un *Utilisateur* peut être abonné à plusieurs *Buffer*.
Ce pattern est composé des classes **Observable**, **Observateur**, **Utilisateur** et **Buffer**.

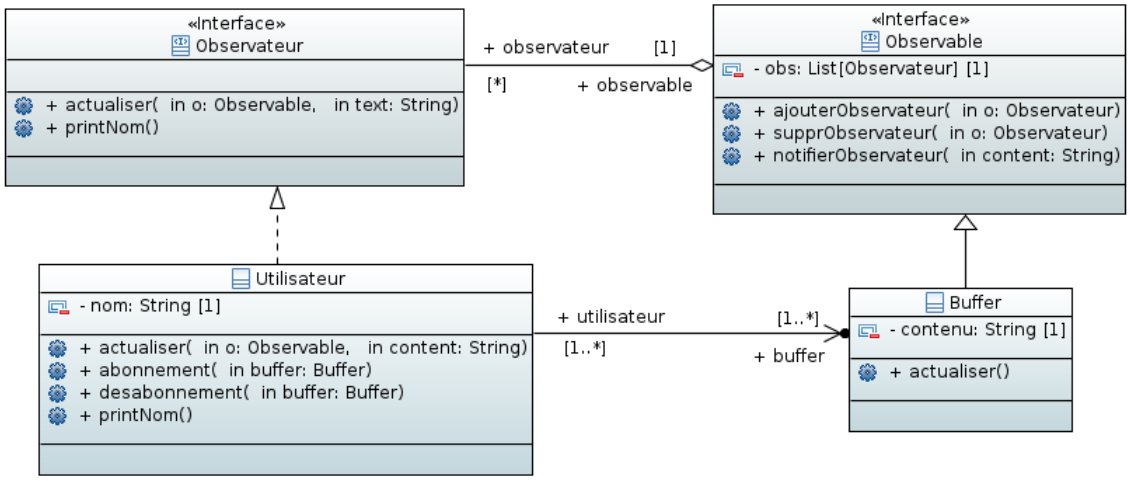

FIGURE 2 – Diagramme UML du patron de conception *Observer*

Le patron de conception Observer permet ici de simuler un partage des données entre plusieurs utilisateurs. Un partage synchronisé en ligne d'un éditeur de texte serait un peu plus complexe mais l'idée reste la même.

4.3 Command & Composite

Toutes les commandes exécutables par les utilisateurs sur notre éditeur ont été implémentées en respectant le patron de conception Command. La possibilité de réaliser des macros quant à elle suit le schéma du patron Composite. Ces deux patterns fonctionnent en paire dans l'éditeur puisqu'il nous fallait une méthode commune et générique à toutes les commandes pour la création de macro qui est une composition de commandes.

Ces patterns sont représentés par les classes **Commande**, **Macro**, **Buffer**, et toutes les classes de **Commandes** basiques de l'éditeur.

Command

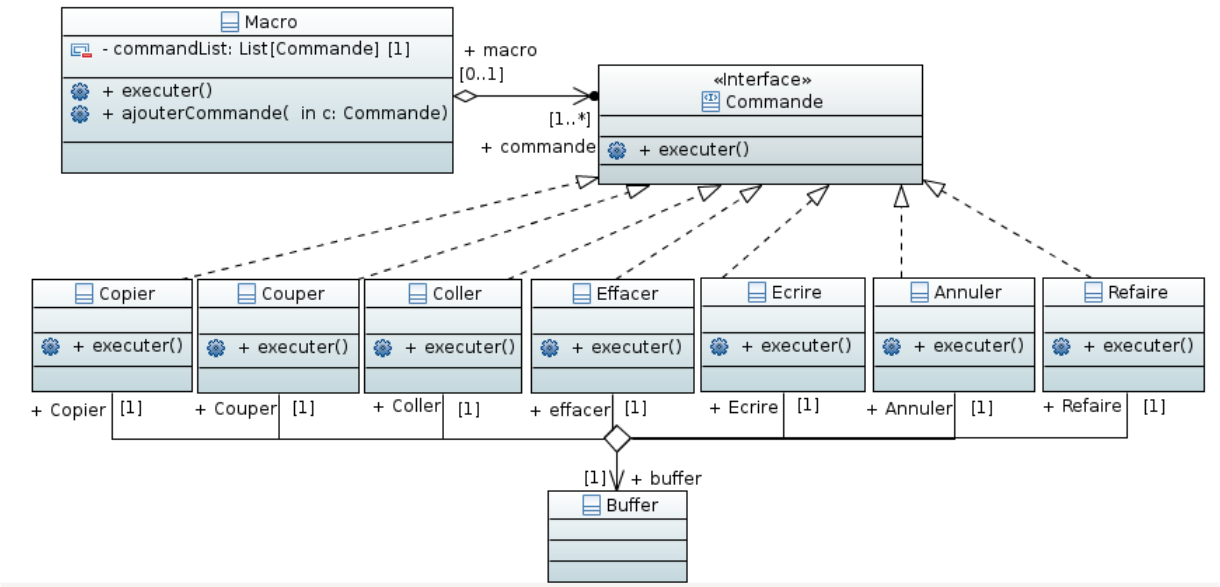

FIGURE 3 – Diagramme UML du patron de conception *Command*

Dans notre projet il existe en réalité deux invocateurs des commandes à savoir **Macro** et **Buffer**. Ce dernier se trouve être également le récepteur.

L'invocateur ne connaît pas le l'intérieur des **Commandes** il leur demande juste de s'exécuter. Il ne connaît pas non plus le récepteur, nous préservons ainsi l'encapsulation.

Composite

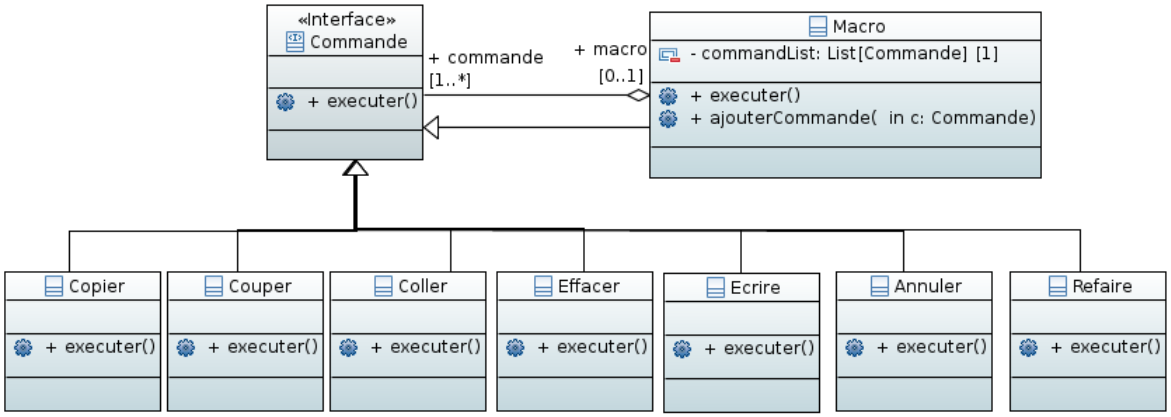

FIGURE 4 – Diagramme UML du patron de conception *Composite*

Toutes les commandes de base de l'éditeur interagissent avec le buffer via une seule méthode générique : *executer()*. En plus de simplifier la compréhension du code et de le rendre plus maintenable, l'implémentation de ce pattern permet la création de macros beaucoup plus facilement. Grâce à cette méthode générique toutes les commandes seront appelées de la même façon.

5 Conclusion

Ce projet nous a permis de nous remémorer les design patterns vus en licence 3, de nous familiariser avec le langage objet Scala et les tests JUnit et d'apprendre à perdre moins de temps sur Papyrus et Maven.

En effet, l'année dernière en troisième année de licence informatique nous avons vu les design patterns lors du module de Programmation Orienté Objet, c'est avec joie que nous les avons retrouvés tout au long de ce projet. L'idée de patron de conception est assez intéressante, ils permettent souvent une simplification et une compréhension universelle du code. Ils sont, en revanche, pour nous encore assez difficile à penser au sein d'un code. Nous ne pensons pas automatiquement à l'implémentation d'un pattern en voyant tels ou tels types de méthodes. Ce projet nous aura aidés à y voir encore un peu plus clair dans ce domaine.

Scala et les JUnit tests sont tout nouveaux pour nous. Nous n'avions jamais eu à nous en servir dans notre jeune passé de développeurs. Scala étant proche du Java fut globalement assez simple à prendre en main, nous n'avons pas eu de difficulté particulière ni de blocage sévère. De même que pour Scala, JUnit dans l'utilisation que nous en avons faite fut facile d'approche.

Là où nous avons en revanche perdu énormément de temps au cours du projet furent les parties en relation avec Papyrus et Maven. Nous n'avions jamais eu à utiliser ces frameworks et nous ne les avons pas trouvé très intuitifs. Nous avons essayé de faire de notre mieux selon ce que nous avons compris.

Ce fut dans l'ensemble un projet vraiment intéressant nous initiant à de nouveaux frameworks et proposant un la conception et l'implémentation d'un logiciel sous un nouvel angle, plus professionel que ce que nous avons fait jusque-là.

6 Annexe

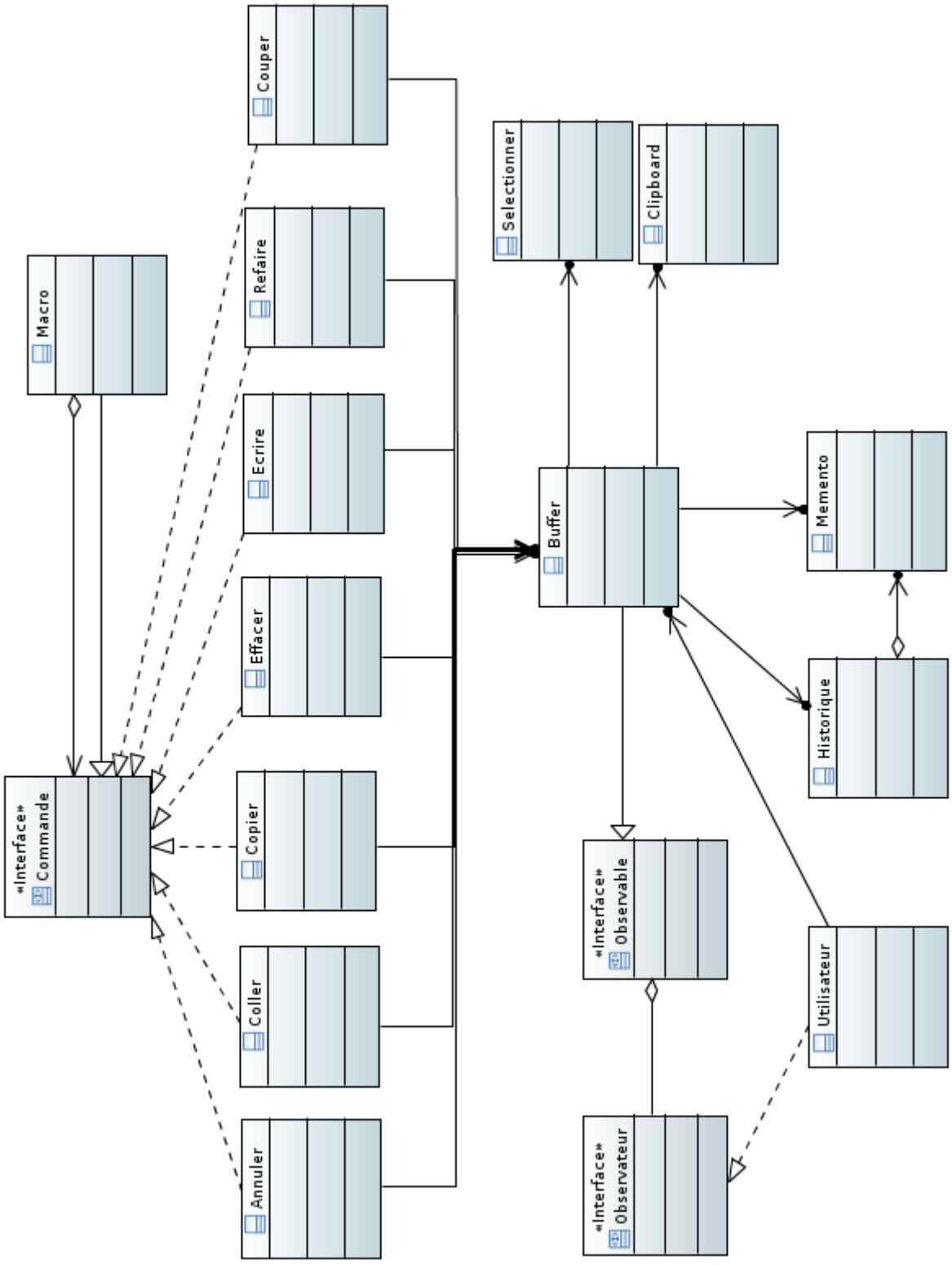

FIGURE 5 – Diagramme UML du mini-éditeur