Université de Nantes
Master 1 ATAL

Projet Génie Logiciel - Version 1

Mini-éditeur de texte

Adeline GRANET,
Alexis LINARD

le 15 novembre 2013

Table des matières

1 Introduction

Le projet *Mini Editeur* a pour but la réalisation d'un outil d'éditeur de texte. Il s'inscrit dans le module de Génie Loiciel. A travers lui, nous apprenons à utiliser différents outils tel que le langage Scala avec sa bibliothèque graphique Swing dans le logiciel Éclipse ce qui nous permet de compiler, déboguer et utiliser github pour la gestion de version.

L'objectif de ce rapport est de présenter la modélisation de notre mini-éditeur de texte tel que nous l'avons conçu tout en intégrant les principaux diagrammes d'UML, certains patrons de conception et ainsi que les techniques liés à l'environnement de travail. Dans une première partie, nous présenterons le projet. L'éditeur sera présenté de manière approfondi avec ses fonctionnalités.

Dans une seconde partie, nous exposerons la vision du projet sous forme de diagramme de classe avec l'explication des patrons de conceptions utilisés.

2 Présentation du projet : le Mini-Editeur

Nous allons présenter dans cette partie l'éditeur de texte dans sa globalité avec les différentes fonctionnalités que l'on devra intégrer ainsi que le contenu de chaque version.

2.1 Le principe

Ce projet reste un éditeur de texte tout ce qu'il y a de plus basique avec les fonctionnalités qu'on lui connait. Certains conceptions et fonctionnalités nous ont été imposé comme suit :

- le texte est contenu dans un Buffer (zone de travail) ;
- il existe une notion de sélection de texte, avec des commandes utilisateur permettant de déplacer le début et la fin de la sélection ;
- copie de la sélection dans le presse-papier ;
- copie de la sélection dans le presse-papier puis effacement de la sélection ;
- remplacement de la sélection par le contenu du presse-papier ;
- l'interface homme-machine est d'un type quelconque (textuelle ou graphique).

Pour ce dernier point, nous avons choisi une version graphique simple pour interagir avec l'utilisateur.

2.2 En plusieurs versions

La conception du logiciel doit se faire en spirale c'est-à-dire fournir des livrables à intervalle donné et enrichir la version précédente à chaque fois. Nous aurons trois version de l'éditeur de texte.

la version 1 contiendra uniquement les actions d'édition de base ainsi que l'interface graphique correspondante ;

la version 2 permettra d'enregistrer les actions de l'utilisateur et de les rejouer ;

la version 3 permettra de réaliser le défaire/refaire, avec une capacité quelconque dans le défaire (c'est-à-dire être capable de revenir à l'état initial)

A chaque étape, nous ferons évoluer l'interface graphique en conséquence de l'ajout/modification des différentes options.

Ce projet est conçu pour que l'on puisse répondre à certains objectifs pédagogique qui ont été fixé comme être capable de présenter une tehnique de conception en utilisant différents patrons de conception ; être capable de faire en parallèle la réalisation de diagramme statique et dynamique en UML.

Dans ce rapport c'est donc la première version que nous allons présenter avec le(s) patron(s) mis en jeu.

3 Implémentation de la version 1

3.1 Patrons de conceptions

Singleton Le but de ce patron de conception est de limiter une classe à une seule et unique instance. Pour notre projet, nous avons choisit de mettre la classe *Buffer* en singleton car plusieurs interfaces hommmes-machines peuvent être relié à un même Buffer mais l'inverse n'est pas possible.

Nous avons donc mis le constructeur de la classe en privé aisni que l'instance de la classe, et une méthode publique qui permet de récupérer cette même instance. Sur la figure **??**, on peut voir au bas de la classe, les deux méthodes sités.

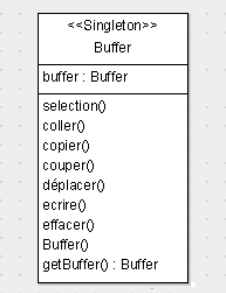

FIGURE 1 – Mise en oeuvre du patron Singleton

Commande Ce patron de conception est de type comportemental. Ce patron va permettre d'encapsuler une action réalisée par l'utilisateur (autrement appellé une requête) dans un objet pour permettre diverses opérations sur la requête. Cela va permettre de dissocier l'objet qui invoque une opération de l'objet qui possède les connaissances nécessaires pour réaliser l'opération demandée. On peut voir l'invocateur comme l'instrument qui appelle la méthode à exécuter. Il est particulièrement recommander en présence d'une IHM car il permet une certaine flexibilité au niveau du design comme le partage de fonctionnalités et le remplacement dynamique des commandes.

Concrètement, nous avons appliqué le patron sur notre projet comme suit :

TABLE 1 – Distribution des rôles

Rôle	**Objet**
Client	IHM
Invocateur	IHM
Commande	Command
CommandeConcrète	Cut, Paste, Copy, Move, Select
Receveur	Buffer

Maintenant que nous avons explicité le rôle de chacun ainsi que les interactions qu'il existe entre eux. Nous vous présentons en figure 2 le diagramme de classe correspondant à notre projet et mettant en œuvre le patron Commande.

Afin d'expliciter un peu plus encore le fonctionnement, nous présentons en figure 3 et 4 des exemples de diagramme de séquence où l'on peut voir les interactions entre chaque classe.

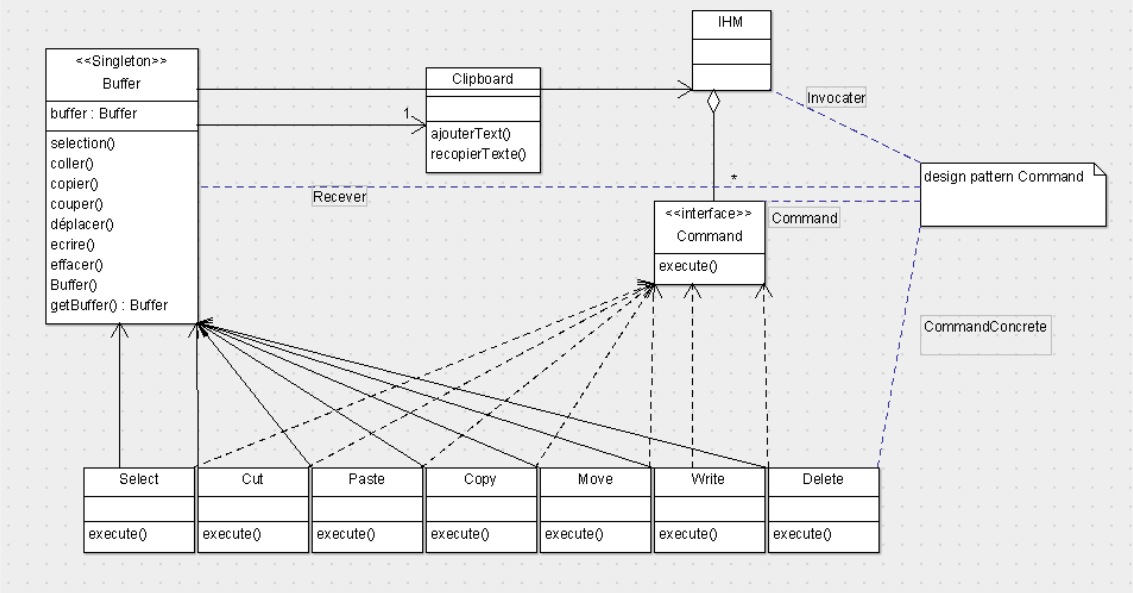

FIGURE 2 – Mise en oeuvre du patron Commande

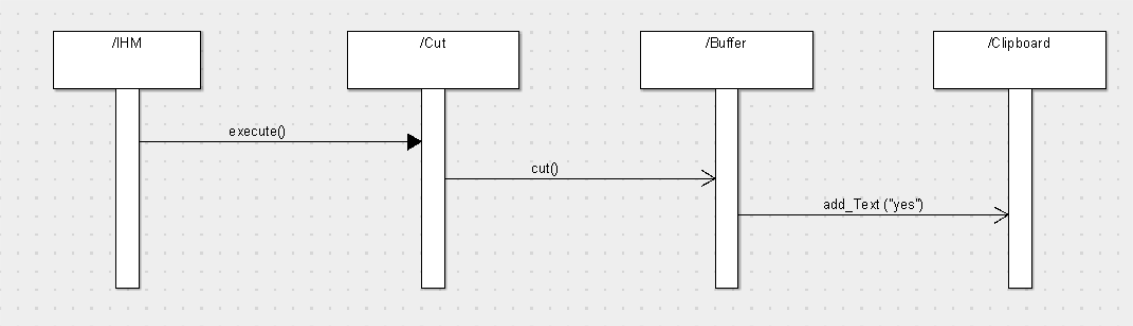

FIGURE 3 – Diagramme de séquence pour couper

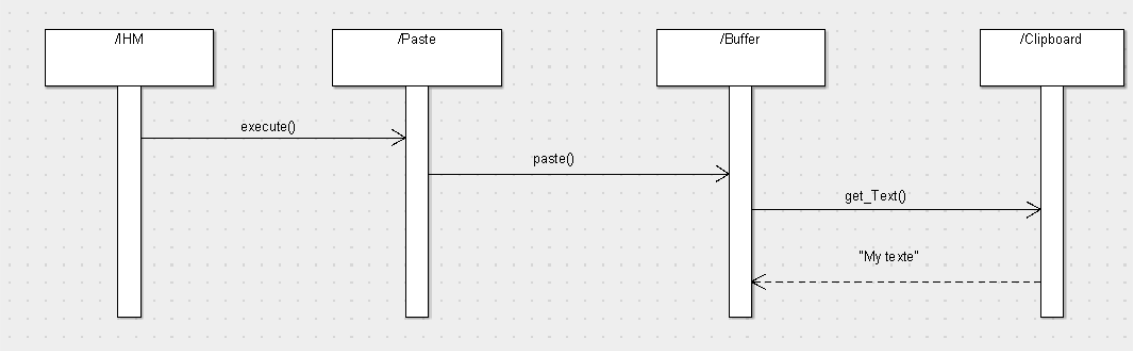

FIGURE 4 – Diagramme de séquence pour coller

3.2 Interface graphique

Nous présentons en figure 6, l'interface que nous avons réalisé pour l'utilisateur. D'un point d evue générale, nous avons deux parties distinctes dans cette interface : la partie éditeur avec l'espace pour écrire du texte et la partie interaction avec les boutons. On y retrouve les différentes fonctions qui nous ont été demandé : copier, coller, couper. Il y a également deux autres boutons qui sont désactivé pour l'instant car ils sont en préparation pour la prochaine version avec le concept *undo redo*. Grâce à Scala, nous n'avons pas eu besoin de créer un bouton *select* car ce

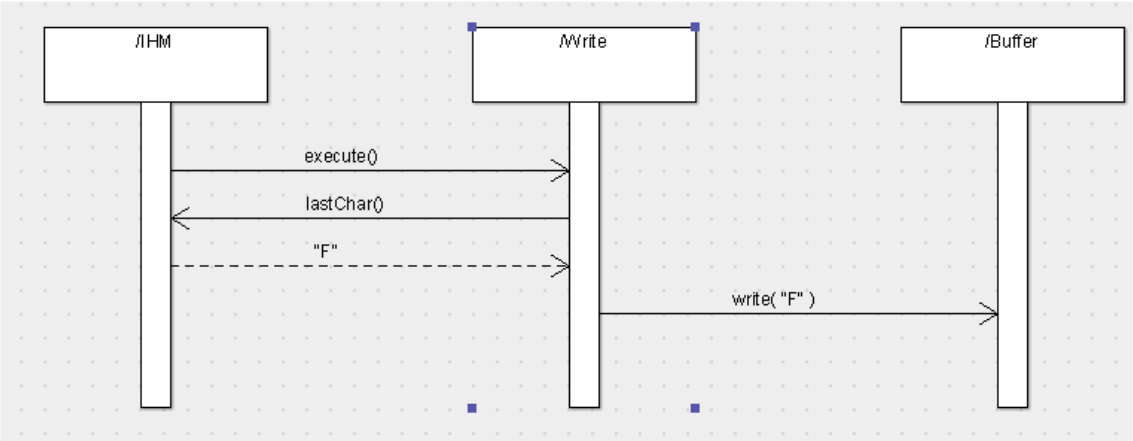

FIGURE 5 – Diagramme de séquence pour écrire

dernier est automatiquement gérer.

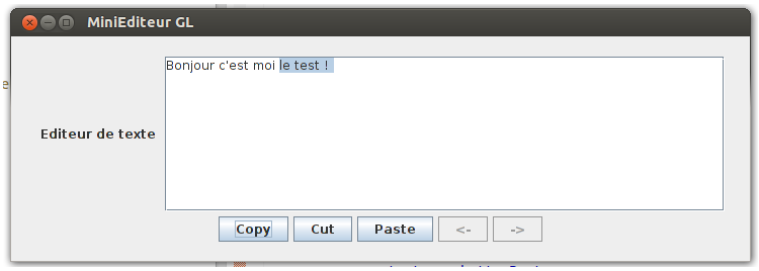

FIGURE 6 – L'interface graphique réalisée

4 Conception Détaillé

4.1 Scala

Scala est un langage de programmation multi-paradigme pour exprimer les modèles de programmation courants dans une forme concise et élégante.

Scala intègre les paradigmes de programmation orientée objet et de programmation fonctionnelle, avec un typage statique. Il concilie ainsi ces deux paradigmes habituellement opposés et offre au développeur la possibilité de choisir le paradigme le plus approprié à son problème. C'est pour cela qu'en tant que développeurs, nous avons l'impression de développer à la fois en Java, et en OCamL !

Ce langage est prévu pour être compilé en bytecode Java (exécutable sur la JVM)

FIGURE 7 – Scala

4.2 Maven

Apache Maven est un outil logiciel libre pour la gestion et l'automatisation de production des projets logiciels Java et Java EE en particulier. L'objectif recherché est comparable au système Make sous Unix, que nous étudions dans le cadre du module *Concepts et Outils de Développement* : produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication. Contrairement à Make et aux Autotools, Maven est plus adapté à Java, et surtout bien plus utilisé dans le monde du travail (Make est pour aisi dire déprécié).

FIGURE 8 – Apache Maven

Maven impose une arborescence et un nommage des fichiers du projet. En voici une liste non exhaustive :

- `/src` : les sources du projet
- `/src/main` : code source et fichiers source principaux
- `/src/main/scala` : code source
- `/src/main/resources` : fichiers de ressources (images, fichiers annexes etc.)
- `/src/main/webapp` : webapp du projet (lorsque projet web - ce n'est pas notre cas)
- `/src/test` : fichiers de test
- `/src/test/scala` : code source de test
- `/src/test/resources` : fichiers de ressources de test
- `/target` : fichiers résultat, les binaires (du code et des tests), les packages générés et les résultats des tests

Maven utilise un paradigme connu sous le nom de Project Object Model (POM) afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production. Il est livré avec un grand nombre de tâches pré-définies, comme :

- `clean` : permet de supprimer le contenu du répertoire /target
- `compile` : permet la compilation du code source
- `test` : permet l'exécution des tests unitaires

– `package` : permet la génération d'une release (génération d'une archive JAR et de la scaladoc)

De plus, l'idée est que, pour n'importe quel but, tous les buts en amont doivent être exécutés sauf s'ils ont déjà été exécutés avec succès et qu'aucun changement n'a été fait dans le projet depuis.

Dans le cadre du projet MiniEditeur, nous utilisons cet outil essentiellement pour la gestion des dépendances, afin d'éviter d'importer les archives .jar dans le classpath "à la main". Voici un extrait du `pom.xml` pour le projet :

Listing 1 – Exemple de fichier pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
            xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>MiniEditeur</groupId>
        <packaging>jar</packaging>
        <artifactId>MiniEditeur</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <name>MiniEditeur</name>
        <description>MiniEditeur projet Genie Logiciel - Adeline Granet and Alexis
            Linard</description>

                ...

        <dependencies>

                ...

                <dependency>
                        <groupId>org.scala-lang</groupId>
                        <artifactId>scala-swing</artifactId>
                        <version>2.11.0-M5</version>
                </dependency>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.11</version>
                </dependency>
        </dependencies>
                ...
</project>
```

4.3 Ajout et génération de scaladoc

Scaladoc est un outil permettant de créer une documentation d'API en format HTML depuis les commentaires présents dans un code source en Scala. Scaladoc est le standard industriel pour la documentation des classes Scala. En outre, cela permet de commenter convenablement le code de notre mini éditeur. Les développeurs utilisent certains styles de commentaire et des tags Scaladoc [1] quand ils documentent un code source. Un bloc de commentaire scala commençant par /**

1. sur le modèle de la Javadoc, son homologue dans le cas de programmes codés en Java

deviendra un bloc de commentaire Scaladoc qui sera inclus dans la documentation du code source. Les commentaires scaladoc précèdent généralement les déclarations de classes, d'attributs et de méthodes. Voici certains tags (liste non exhaustive), et un exemple mis en pratique dans le projet MiniEditeur :

- `@author` : Nom du développeur
- `@deprecated` : Marque la méthode comme dépréciée. Certains IDEs créent un avertissement à la compilation si la méthode est appelée.
- `@exception` : Documente une exception lancée par une méthode — voir aussi `@throws`
- `@param` : Définit un paramètre de méthode. Requis pour chaque paramètre.
- `@return` : Documente la valeur de retour. Ce tag ne devrait pas être employé pour des constructeurs ou des méthodes définis avec un type de retour void.
- `@see` : Documente une association à une autre méthode ou classe.
- `@since` : Précise à quelle version de la SDK/JDK une méthode a été ajoutée à la classe.
- `@throws` : Documente une exception lancée par une méthode.
- `@version` : Donne la version d'une classe ou d'une méthode.

Listing 2 – Exemple de scaladoc

```
/**
 *
 * Class containing the User Interface
 * @author AdelineAlex
 * @todo une belle IHM dans une webapp peut tre
 *
 */
object UserInterface extends SimpleSwingApplication {

        /**
         * Variable containing the copied value in the text editor
         */
        var copiedValue: String = ""

        /**
         * @TODO For Saved Actions
         */
        var debut: Int = 0

        /**
         * @TODO For Saved Actions
         */
        var fin: Int = 0

        /**
         * The UI
         */
        def top = new MainFrame {

                //Title of the frame
                title = "MiniEditeur GL"
                        ...
        }
}
```

4.4 Création d'un repository GitHub et utilisation de GIT

Dans le but de permettre le développement collaboratif de notre application, nous avons crée un dépôt GitHub pour l'hébergement du projet. Il repose sur *GIT*, qui est un logiciel de gestion de versions.

Voici le lien pour le projet :

`https://github.com/allinard/miniEditeurGL`

Les commandes que nous avons le plus utilisé pour le moment sont les suivantes :
- `git init` crée un nouveau dépôt.
- `git clone` clone un dépôt distant.
- `git add` ajoute de nouveaux fichiers au repository.
- `git commit` valider des changements dans un fichier. N'envoie pas sur le repository distant.
- `git branch` crée une nouvelle branche de développement.
- `git merge` fusionne plusieurs branches de développement.

– `git pull` récupérer la version la plus à jour du projet depuis le repository distant.
– `git push` envoyer ses changements sur le repository distant.

4.4.1 Virtualisation du projet

Dans un souci d'efficience maximum, nous avons décidé de travailler sur des machines virtuelles [2]. L'avantage de cette pratique est la portabilité de la VM d'un hôte à l'autre. Ainsi, une fois le projet correctement configuré (avec un bon IDE Eclipse bien configuré, et les bons plug-in par exemple), il suffit de la copier d'un collaborateur au projet à un autre. C'est ainsi un bon moyen de ne pas perdre de temps, et de configurer le projet une seule fois.

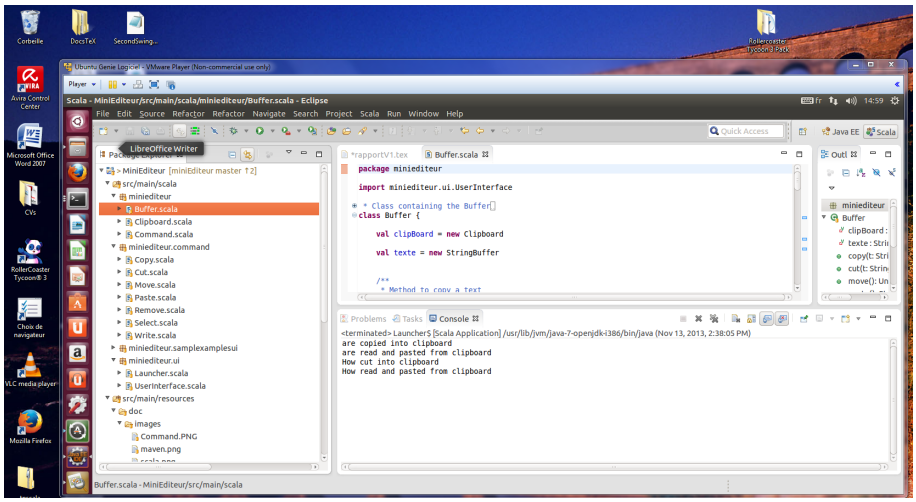

FIGURE 9 – Exécution de la VM de développement OPENDATAWRAPPER

Cependant, un des inconvénients de l'utilisation des machines virtuelles est le besoin de ressources convenables venant de l'hôte. Dans le cas de notre VM :
– 3Go RAM (at least)
– 2 cores
– + or - 10Go Hard Drive

5 Tests Unitaires

Lors du développement d'un logiciel, on peut facilement faire des modifications sans se rendre compte qu'elles introduisent des bugs dans certains cas particuliers : les régressions. Une régression est l'introduction de nouveaux bugs à la suite d'une évolution fonctionnelle. Hors, il n'est pas humainement possible de tester systématiquement tous les cas d'utilisation possible, ces bugs peuvent se retrouver déployés en production, avec tous les problèmes que cela comporte.

Les tests unitaires servent également à cadrer le développement, puisque dans la mesure du possible ils doivent être écrits avant de coder : il s'agit du TDD [3].

Concrètement, les tests unitaires consistent en un ensemble de scripts, qui ont chacun en charge la validation d'un morceau de code (ou d'une classe dans le cadre de la programmation orientée

2. En informatique, une machine virtuelle (anglais virtual machine, abr. VM) est une illusion d'un appareil informatique créée par un logiciel d'émulation. Le logiciel d'émulation simule la présence de ressources matérielles et logicielles telles que la mémoire, le processeur, le disque dur, voire le système d'exploitation et les pilotes, permettant d'exécuter des programmes dans les mêmes conditions que celles de la machine simulée

3. Technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel. Ces tests permettent aussi de préciser les spécifications du code et donc son comportement ultérieur en fonction des situations auxquelles il sera exposé. Ce qui facilite la production d'un code valide en toutes circonstances. On obtient donc un code plus juste et plus fiable.

objet). Il existe de très nombreux frameworks : JUnit (Java), SimpleTest (PHP), CppUnit (C++), JSUnit (Javascript), unittest (Python), NUnit (.NET), etc.

Nous avons réalisé nos tests unitaires avec JUnit, qui est décrit ci-dessous. Cependant, nous connaissons également ScalaTest, que nous n'avons pas encore mis en place, ce que nous ferons peut être dans les prochaines versions du mini éditeur...

5.1 JUnit

JUnit est un framework de test unitaire initialement développé pour le langage de programmation Java, et qui possède des extensions pour le langage Scala.

JUnit définit deux types de fichiers de tests. Les TestCase sont des classes contenant un certain nombre de méthodes de tests. Un TestCase sert généralement à tester le bon fonctionnement d'une classe. Une TestSuite permet d'exécuter un certain nombre de TestCase déjà définis.

Enfin, JUnit est intégré par défaut dans les environnements de développement intégré Java tels que BlueJ, Eclipse et NetBeans. C'est donc une constante dans le monde de l'entreprise et du développement logiciel !

5.2 La réalisation concrète de tests unitaires avec Scala

Voici un exemple simple de test unitaire en Scala :

Listing 3 – Exemple de TU Scala

```
import junit.framework._
import org.junit.Assert._

/**
 * @author AdelineAlexis
 */
class SampleScalaUnitTest extends TestCase {

	/**
	 * Test de la mthode testOne()
	 * @return OK
	 */
	def testOne() = {
		assertEquals(1, 1)
	}

	/**
	 * Test de la mthode testTwo()
	 * @return FAIL
	 */
	def testTwo() = {
		assertEquals(2, 3)
	}

}
```

6 Conclusion

Cette première modélisation que nous avons réalisé pour l'éditeur de texte est une réponse au cahier des charges qui nous a été fourni. Grâce à cela, nous avons pu définir et mettre en place les principales fonctionnalités que l'éditeur doit posséder.

La réalisation du diagramme de classe, nous a permis de manipuler le patron de conception que l'on a abordé en cours et de le mettre en œuvre. Les diagrammes de séquences nous ont permit de mieux comprendre et analyser comment les différents objets de l'éditeur pouvaient interagir entre eux lors d'une action.

Nous avons également pu réfléchir sur l'agencement de l'interface utilisateur pour qu'elle reste le plus simple possible pour ce dernier tout en restant le plus fonctionnel possible. Tout en sachant que nous devrons l'adapter à chaque version du projet.

La prochaine version devra permettre d'enregistrer les actions de l'utilisateur et de les rejouer.

Table des figures

Liste des tableaux

Références

[1] E. Rudolph, P.Graubmann, and J. Grabowski. Tutorial on message sequence charts (MSC'96). In *FORTE*, 1996.

[2] ITU-TS. ITU-TS Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1997.

[3] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion.* Addison-Wesley, 1997.

[4] L. Lamport. *LaTeX—A Document Preparation System—User's Guide and Reference Manual.* Adsison-Wesley, 2nd edition, 1994. Updated for LaTeX 2_ε.

[5] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—Reference Manual—Describing version* , June 2002. Included in MSC macro package distribution.

[6] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—User Manual—Describing version* , June 2002. Included in MSC macro package distribution.

[7] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—Maintenance document—Describing version* , June 2002. Included in MSC macro package distribution.