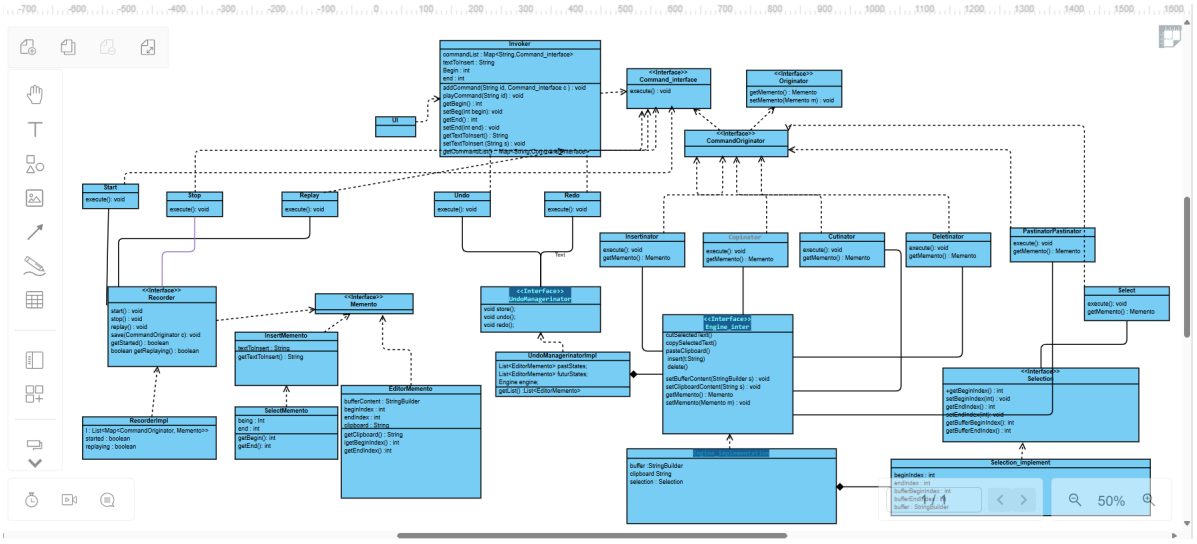
Choix personnelles de conceptions/test

Conception :

Pour la conception de ce projet, j'ai suivi le diagramme UML ci-contre.



Je tiens à m'excuser pour la qualité visuelle de ce diagramme, car je pensais initialement que le site que j'utilisais était gratuit, mais il s'est avéré que ce n'était pas le cas, ce qui a limité les options de création visuelle (je l'ai remis en fin de rapport en format paysage)

Quant aux choix de conception, il faut avouer que je n'en ai pas eu beaucoup à faire. En réalité, le seul choix véritablement important que j'ai eu à faire concerne la version 3 du projet (V3). J'ai opté pour la solution 1, c'est-à-dire la sauvegarde des états du TextEditor. Ce choix a été effectué par un processus d'élimination. La solution 4, qui n'avait apparemment jamais été tentée, me semblait au-dessus de mes compétences et a donc été écartée. La solution 3 m'a paru difficile à comprendre, donc elle a aussi été rapidement abandonnée. Il ne me restait plus que la solution 1 et la solution 2 à comparer. La première me semblait la plus logique, car elle ressemblait fortement à ce que nous avions déjà réalisé pour la version 2 du projet (V2). Elle semblait donc être la solution la plus simple et la plus adaptée à mes capacités.

Concernant l'interface graphique, j'avais prévu d'utiliser JavaFX, mais cette technologie m'a malheureusement abandonnée au moment où j'en avais le plus besoin. Avec le temps limité dont je disposais, je n'avais pas les ressources nécessaires pour me former rapidement à Swing, une autre bibliothèque graphique. Ainsi, j'ai opté pour une interface très basique, fonctionnant directement en ligne de commande via des entrées console. Bien que cela ait été une solution de secours, elle a permis de réaliser les fonctionnalités essentielles du projet sans trop compromettre son efficacité.

Tests :

En ce qui concerne les tests, au moment où j'écris ce rapport, je dois admettre que je n'ai pas pu en réaliser autant que j'en avais prévu au départ. Mon objectif initial était de tester chaque aspect du projet afin de m'assurer que le TextEditor fonctionnerait correctement dans toutes les situations possibles. Cela inclut la gestion des cas les plus divers, comme les entrées nulles, les dépassements de bornes, mais aussi des scénarios plus classiques comme les opérations de "insert", "delete", "undo", et ainsi de suite. L'idée était de couvrir le maximum de possibilités pour garantir que le programme puisse gérer toutes les situations de manière fluide et sans erreur. Cependant, face au manque de temps et de main-d'œuvre (c'est-à-dire, avec peu de personnes pour m'aider), je n'ai pas pu tester l'intégralité des fonctionnalités du programme, ce qui m'a obligée à faire des choix et à me concentrer sur les classes les plus cruciales du projet.

Voici les principales classes que j'ai priorisées pour les tests :

- Engine_implementation : Cette classe est essentielle car elle gère la majeure partie des commandes du TextEditor. Toute la logique de manipulation du texte repose sur elle, il était donc primordial de tester son bon fonctionnement pour m'assurer que les commandes de base comme l'insertion ou la suppression de texte sont bien gérées.

- Selection_implementation : La gestion de la sélection de texte est au cœur du bon fonctionnement de l'éditeur. Si cette fonctionnalité ne fonctionne pas correctement, il devient impossible de gérer le texte de manière efficace. C'est pourquoi j'ai porté une attention particulière à cette classe, car elle est directement liée à l'expérience de l'utilisatrice.

- UndoManager : La fonctionnalité "undo/redo" est cruciale dans un éditeur de texte moderne, car elle permet à l'utilisatrice de revenir en arrière ou de répéter ses actions. La classe UndoManager permet de gérer ces opérations, ce qui en fait un élément clé à tester pour garantir une bonne manipulation des actions.

- Recorder : Cette classe, qui est chargée d'enregistrer les actions effectuées, m'a aussi semblé importante à tester. Cependant, avec du recul, je pense que j'aurais dû me concentrer davantage sur la classe Memento, car elle joue un rôle plus central dans la gestion de l'historique des actions, notamment pour le mécanisme d'"undo/redo". Recorder aurait mérité une attention moindre comparée à Memento, car cette dernière gère les snapshots des états du TextEditor, ce qui est essentiel pour les fonctionnalités d'annulation.

- Invoker : C'est une autre classe très importante, car elle fait le lien direct entre l'utilisatrice et les autres classes du projet. Elle est responsable de l'exécution des

commandes que l'utilisatrice choisit. Tester cette classe m'a permis de m'assurer que l'interface entre l'utilisatrice et le programme est fluide et fonctionne comme prévu.

En plus de ces classes spécifiques, j'ai également veillé à tester toutes les commandes de base de la version 1. Ces fonctionnalités constituent la base même de l'éditeur de texte, et sans une version stable et fonctionnelle de cette première version, il m'aurait été impossible d'implémenter correctement les fonctionnalités des versions 2 et 3. Tester la version 1 a donc été une étape indispensable pour m'assurer que l'ensemble du projet pourrait évoluer sans causer de régressions majeures.

Cependant, au fil de mon travail et après en avoir discuté avec certaines camarades, il m'a été suggéré qu'une approche plus rigoureuse en Test Driven Development (TDD) aurait sans doute été plus bénéfique. En adoptant cette méthode, j'aurais pu rédiger les tests avant même de commencer à coder les fonctions, ce qui m'aurait permis de tester les fonctionnalités au fur et à mesure de leur développement. Cette approche aurait facilité une couverture de tests plus complète et m'aurait permis de m'assurer que chaque fonction soit bien testée avant d'être intégrée. De plus, cela m'aurait permis de mieux suivre l'évolution du projet et de m'assurer que le code reste propre et fonctionnel tout au long du développement. En pratiquant le TDD, j'aurais également évité certains problèmes de dernière minute, en identifiant les bugs plus tôt dans le processus de développement.

Bref, même si je n'ai pas pu tester tous les aspects du projet comme prévu, je pense avoir couvert les parties les plus cruciales du code. Mais je suis consciente qu'il me reste encore beaucoup de tests à effectuer pour m'assurer que le projet est complet et robuste. Avec un peu plus de temps, je pourrai tester de manière plus approfondie et rendre l'application encore plus fiable.

Synthèse des résultats des tests :

À cause du manque de temps, je n'ai pas pu faire autant de tests que je l'aurais voulu. Du coup, la synthèse des résultats n'est peut-être pas totalement représentative de la réalité. Mais d'après les tests que j'ai pu faire, la version 3 semble fonctionner correctement dans les situations que j'ai testées. Tous les tests sont passés, donc on peut dire que la V3 est assez solide dans les cas que j'ai pu envisager.

Cela dit, il faut quand même noter que je n'ai testé qu'un petit nombre de classes, moins de dix, et chaque classe n'a pas été testée en profondeur. Par conséquent, certains cas moins courants ou plus spécifiques n'ont probablement pas été pris en compte, et c'est là qu'il pourrait y avoir des problèmes si ces situations inattendues arrivent. Même si le programme semble stable dans les tests réalisés, il reste des zones d'ombre qui devront être vérifiées avec plus de tests.

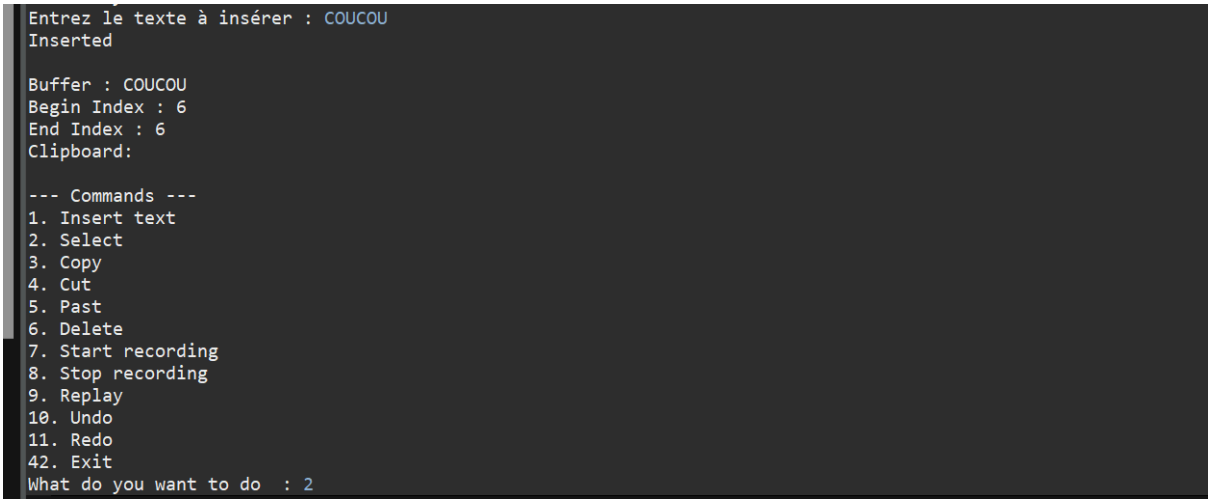
Comme je suis en train de rédiger ce rapport et que je vais continuer à travailler dessus, j'espère avoir le temps de compléter les tests. Mon plan est de rajouter les classes manquantes et de tester un peu plus en profondeur, idéalement avec au moins deux tests par classe. Cela me permettra de m'assurer que tout fonctionne bien et de mieux couvrir les différents cas possibles.

Comment lancer les différents versions

Comment lancer les différentes versions :
Malheureusement, il n'y a pas de versions distinctes pour cet éditeur de texte, car la version 2 (V2) et la version 3 (V3) apportent des fonctionnalités supplémentaires tout en conservant les mêmes bases que la version 1 (V1). Par conséquent, le processus de lancement reste le même pour les trois versions.

Pour exécuter le programme, il suffit d'appuyer sur "Run" dans votre environnement de développement. Une fois l'application lancée, une série de 10 propositions sera affichée dans la console. Vous devrez entrer le chiffre correspondant à l'action que vous souhaitez réaliser. Voici un aperçu des actions disponibles :

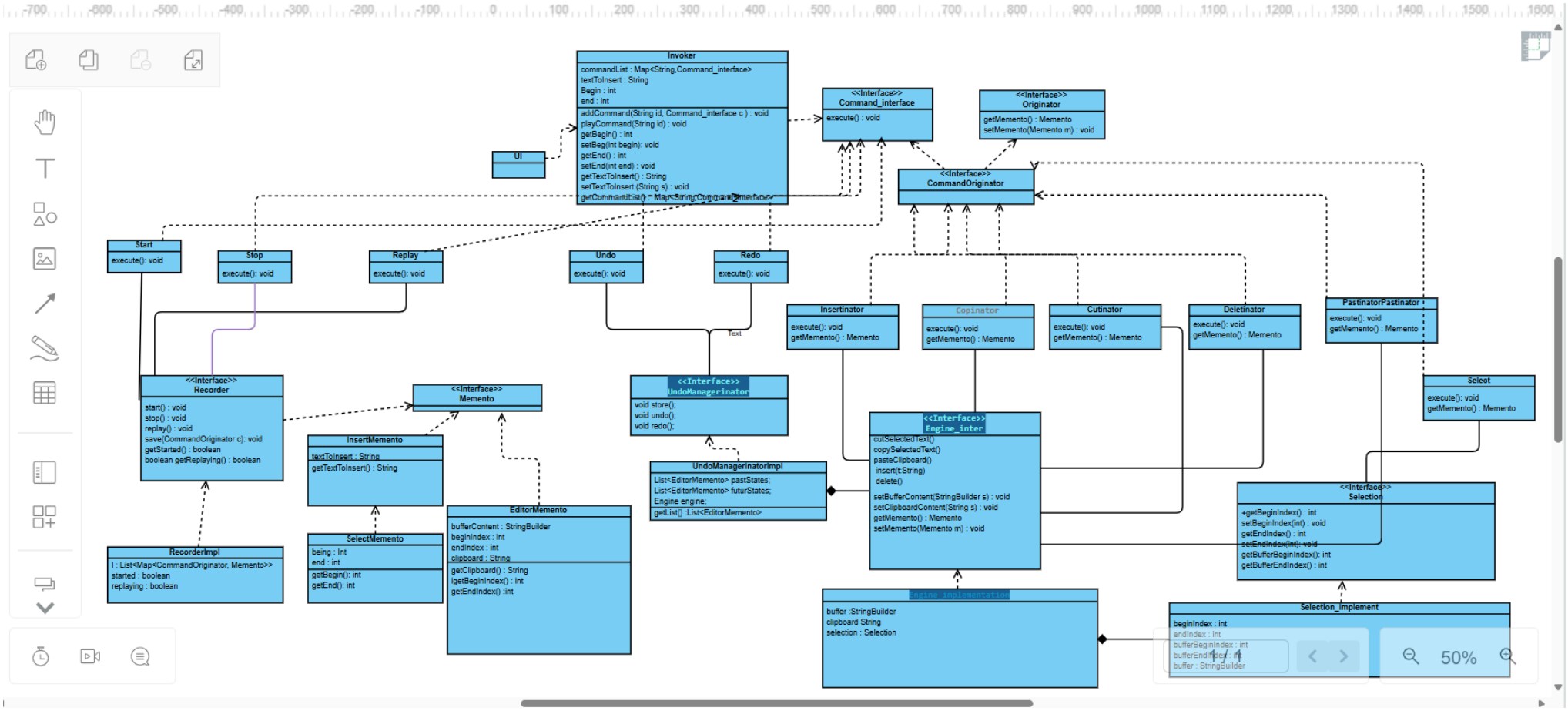
- Inserer : Le mini-éditeur vous demandera de saisir un texte à insérer. Une fois l'opération réalisée, le texte apparaît dans le buffer au-dessus des propositions.
- Select : Vous devrez indiquer l'indice de début et de fin pour la sélection de texte. Celle-ci sera mise à jour en conséquence, toujours visible au-dessus des propositions.
- Copy : Cette action copiera la sélection dans le presse-papiers.
- Cut : Elle coupera le texte sélectionné.
- Paste : Cette action collera le contenu du presse-papiers à l'endroit spécifié.
- Delete : Elle supprimera le texte sélectionné.
- Replay : Vous permettra de rejouer les actions enregistrées.
- Undo : Annule la dernière action effectuée. Pour des raisons obscures il faut parfois appuyer plusieurs fois pour que la commande fonctionne
- Redo : Refait l'action qui avait été annulée.
- Exit : Quitte l'éditeur de texte.



Voici un petit aperçu de ce que cela donne.

Remerciement

Je tiens à remercier SAVI Oumar qui m'a aidé lors de la création de l'UML et la conception ainsi que HOANG Amaya qui m'a aidé pour faire l'interface graphique.


Invoker
commandList : Map<String,Command_interface>
textToInsert : String
Begin : int
end : int
addCommand(String id, Command_interface c) : void
playCommand(String id) : void
getBegin() : int
setBeg(int begin): void
getEnd() : int
setEnd(int end) : void
getTextToInsert() : String
setTextToInsert (String s) : void
UI
<<Interface>>
Command_interface
execute() : void
<<Interface>>
Originator
getMemento() : Memento
setMemento(Memento m) : void
<<Interface>>
CommandOriginator
Start
execute(): void
Stop
execute(): void
Replay
execute(): void
Undo
execute(): void
Redo
execute(): void
Insertinator
execute(): void
getMemento() : Memento
Copinator
execute(): void
getMemento() : Memento
Cutinator
execute(): void
getMemento() : Memento
Deletinator
execute(): void
getMemento() : Memento
PastinatorPastinator
execute(): void
getMemento() : Memento
Select
execute(): void
getMemento() : Memento
<<Interface>>
Recorder
start() : void
stop() : void
replay() : void
save(CommandOriginator c): void
getStarted() : boolean
boolean getReplaying() : boolean
<<Interface>>
Memento
InsertMemento
textToInsert : String
getTextToInsert() : String
SelectMemento
being : Int
end : int
getBegin(): int
getEnd(): int
EditorMemento
bufferContent : StringBuilder
beginIndex : int
endIndex : int
clipboard : String
getClipboard() : String
igetBeginIndex() : int
getEndIndex() :int
<<Interface>>
UndoManagerinator
void store();
void undo();
void redo();
Text
UndoManagerinatorImpl
List<EditorMemento> pastStates;
List<EditorMemento> futurStates;
Engine engine;
getList() :List<EditorMemento>
<<Interface>>
Engine_inter
cutSelectedText()
copySelectedText()
pasteClipboard()
insert(t:String)
delete()
setBufferContent(StringBuilder s) : void
setClipboardContent(String s) : void
getMemento() : Memento
setMemento(Memento m) : void
<<Interface>>
Selection
+getBeginIndex() : int
setBeginIndex(int) : void
getEndIndex() : int
setEndIndex(int): void
getBufferBeginIndex(): int
getBufferEndIndex() : int
buffer :StringBuilder
clipboard String
selection : Selection
Selection_implement
beginIndex : int
endIndex : int
bufferBeginIndex : int
bufferEndIndex : int
buffer : StringBuilder
RecorderImpl
l : List<Map<CommandOriginator, Memento>>
started : boolean
replaying : boolean
50%