Raphaël FROMENTIN
Alice POINSATTE
Jordan DIZIN
Lili-Rose GICQUEL
Gloddy NGOKOLI

Rapport Projet

Chromat-Ynk

Mohamed Haddache

Java 2023-2024

Sommaire

Introduction1

1. Introduction

Chromat'ynk est un langage de programmation écrit en Java qui permet de dessiner des formes quelconque sur un canvas à partir de diverses instructions. Nous pouvons avancer ou bien reculer d'un certain nombre de pixels, tourner d'un certain angle, choisir l'épaisseur du trait, sa couleur… Notre projet comprend le langage de programmation ainsi qu'un éditeur graphique fait avec JavaFX.

2. Gestion de projet

2.1. Mise en place

Après la réunion d'information du lundi 06/05, nous nous sommes réunis pour mettre en place les fondations du projet. Nous avons commencé par créer un nouveau projet Maven en utilisant l'archétype officiel de JavaFX. Nous y avons ajouté un fichier .gitignore puis nous l'avons hébergé sur un dépôt GitHub. Jordan a ensuite ajouté une intégration continue au projet. Il s'agit d'une Action GitHub qui se lance à chaque commit sur le dépôt afin de vérifier si le code compile et (implémenté plus tard) s'il vérifie tous les tests.

2.2. Répartition des tâches

Dès lundi 06/05, nous nous sommes réunis pour mettre en place le projet et lister les tâches à faire sous la forme d'issues sur GitHub. Nous avons utilisé le système de labels de la plateforme pour catégoriser les différentes tâches à faire (bug, nouvelle fonctionnalité etc…). Les issues nous permettent aussi d'assigner un membre à une tâche pour éviter que deux personnes travaillent sur le même problème. Voici la répartition des tâches entre les membres:

- Alice
  - AST (Abstract Syntax Tree)
  - Vérification types
- Gloddy
  - Editeur de code
  - Opérateur modulo
  - Polissage des autres parties

- Jordan
  - Editeur code
  - Intégration continue
  - Ajout d'exemples
  - README
- Lili-Rose
  - Bytecode
  - Compilation
  - Interprétation (VM)
- Raphaël
  - Parsing (analyse lexicale et syntaxique)
  - Editeur code
  - Mise en place du projet (Maven, JUnit)
  - Aide aux autres membres du groupe

2.3. Gestion des conflits

Nous avons tiré partie du système de branches de Git et de pull requests de GitHub pour pouvoir développer différentes fonctionnalités/corrections de bug en parallèle sans entraver les autres membres du groupe. Une fois la fonctionnalité implémentée/le bug corrigé, une pull request est ouverte pour fusionner les modifications avec la branche principale si celles-ci sont approuvées par un pair et passent les tests (intégration continue).

3. Analyse du projet

3.1. Gestion des dépendances

Notre projet dépend de plusieurs bibliothèques Java. Nous avions tout d'abord besoin de JavaFX puisqu'imposé dans l'énoncé puis nous avons décidé d'inclure JUnit pour l'ajout de tests unitaires et RichTextFX pour pouvoir styliser les différents caractères de l'éditeur de code indépendamment notamment pour y ajouter de la coloration syntaxique. Pour gérer ces différentes dépendances entre les machines des différents membres, nous avons opté, comme mentionné précédemment, pour l'outil de gestion et d'automatisation de projet (build tool) Maven qui nous permet d'avoir des build reproductibles quelque soit la machine ou le système d'exploitation. Maven nous

permet aussi de sélectionner la version de Java adaptée à nos besoins. Nous avons choisi Java 21 pour pouvoir profiter de l'association records + interfaces scellées introduits en Java 14 avec le pattern matching amélioré en Java 21, trois fonctionnalités très utiles dans la manipulation d'arbres comme dans le cadre d'un langage de programmation.

3.2. Méthodes de développement

Pour réaliser notre projet, nous avons combiné deux approches: le Domain Driven Development (DDD) et le Test Driven Development (TDD). La première méthode consiste à représenter fidèlement les données manipulées avant d'écrire la logique. Cela permet de séparer la logique métier de la logique technique, d'éviter de nombreux bug dûs à une mauvaise représentation et de plus facilement tester notre programme. Dans le cadre de notre projet, cela ce traduit par modéliser les différentes représentations de notre programme (code source, tokens, AST, bytecode…) avant de faire les logiques d'analyse et de transition (lexing, parsing, vérification de type…). Le TDD consiste à traduire le cahier des charges en tests unitaires puis d'en écrire l'implémentation. Cette approche est adaptée aux projets qui possèdent des attentes et un cahier des charges clair comme celui-ci.

3.3. Architecture du projet

Avant d'être traduit en dessin, le code source tel qu'entré par l'utilisateur passe par plusieurs étapes (souvent appelées "phases" dans la littérature liée aux langages de programmation) ainsi que plusieurs représentations.

L'exemple suivant va être utilisé pour illustrer les différentes représentations.

```
INT x = 4 + 5
```

3.3.1. Analyse lexicale

L'analyse lexicale consiste à représenter le code écrit par l'utilisateur en une suite de mots et de symboles appelés tokens. Le but est de donner une signification lexicale à chaque terme et repérer ceux qui sont invalides.

Voici le résultat de l'analyse lexicale de l'exemple:

```
new Identifier("INT"), new Identifier("x"), new Assign(), new
LiteralInt(4), new Plus(), new LiteralInt(5)
```

3.3.2. Analyse syntaxique

En linguistique, si l'analyse lexicale peut-être comparée à l'association de chaque mot à une nature, l'analyse syntaxique permet de former une phrase. Ici, la nature de chaque terme est un token et la "phrase" à construire l'AST. L'AST (Abstract Syntax Tree), ou Arbre de Syntaxe Abstraite en Français, est un arbre qui représente le code tapé par l'utilisateur. Cette forme est facile à analyser comme pour faire de la vérification de types ou à transformer en une autre représentation (aka compiler).

L'AST obtenu depuis la suite de tokens ci-dessus est le suivant:

```
new DeclareVariable(
  Type.INT,
  "x",
  new Add(new LiteralInt(4), new LiteralInt(5))
)
```

3.3.3. Vérification de types

Notre langage dispose d'un typage statique. Cela veut dire que le type de chaque variable et chaque valeur est connu avant l'exécution du programme. Cela nous permet de vérifier leur cohérence, ce qu'on appelle la vérification de type ou "type checking" en Anglais. Le système de vérification remplit donc deux tâches:

- Déterminer le type de chaque expression
- Vérification que ce type soit cohérent avec l'utilisation de la dite expression

En reprenant l'AST ci-dessus, on constate que `new LiteralInt(4)` et `new LiteralInt(5)` sont de type `Type.INT`, alors l'expression suivante:

```
new Add(new LiteralInt(4), new LiteralInt(5))
```

est également de type `Type.INT`. Enfin, puisque la variable `x` est déclarée comme ayant le type `Type.INT` (premier argument dans l'AST) alors ce programme est cohérent.

Voici un exemple d'AST ne passant pas la vérification de type:

```
new Add(new LiteralInt(4), new LiteralBool(true))
```

La vérification de type nous sert en même temps à vérifier l'existence des variables utilisées. Le code suivant ne passe pas non plus la vérification de type:

```
new DeclareVariable(
  Type.INT,
  "x",
  new Add(new LiteralInt(4), new LiteralInt(5))
),
new VarCall("y") //Appel de la variable "y"
```

3.3.4. Compilation

Pour exécuter notre AST, nous avons choisi de le compiler en une suite d'instructions atomiques qu'on appelle "bytecode". Il s'agit d'un fonctionnement similaire au compilateur Javac ou encore à ce que fait l'interpréteur officiel Python en interne avant exécution du programme.

L'AST suivant:

```
new DeclareVariable(
  Type.INT,
  "x",
  new Add(new LiteralInt(4), new LiteralInt(5))
)
```

est compilé en:

```
new Push(4),
new Push(5),
new Add(),
new Declare(Type.INT, "x")
```

Cette représentation permet de généraliser différentes syntaxes sous un même ensemble d'instructions. Par exemple l'AST suivant:

```
new DeclareVariable(
  Type.INT,
  "x",
  Optional.empty() //Aucune valeur d'initialisation
)
```

est compilé en:

```
new Push(0),
new Declare(Type.INT, "x")
```

3.3.5. Interprétation

L'interprétation consiste à lire chaque bytecode un à un et faire une action en conséquence. Nous avons implémenté cette étape sous forme d'une VM (Virtual Machine) basée sur une pile d'exécution. La plupart des instructions dépilent et/ou empilent des valeurs à l'exécution.

Voici l'exécution pas à pas de cette séquence d'instructions issue des exemples ci-dessus.

```
new Push(4),
new Push(5),
new Add(),
new Declare(Type.INT, "x")
```

`Push(4)`

	4

`Push(5)`

5	4

`Declare(Type.INT, "x")`

La valeur de `x` est `9`.

3.3.6. Horloge d'exécution

Nous devons nous prémunir des potentielles boucles infinies qui peuvent être écrites par l'utilisateur. De plus l'énoncé impose un mode pas à pas ainsi que la possibilité de régler la vitesse d'exécution. Ces trois problèmes sont gérés par la même partie du programme: l'horloge. L'horloge d'exécution (`Clock` dans le projet) permet d'arrêter l'exécution en cours pour la reprendre plus tard. Le projet de cinq types d'horloge:

- `ForeverClock`: n'interrompt jamais l'exécution

- `PeriodClock`: s'interrompt après la prochaine instruction de dessin et ne continue pas malgré les relances du programme tant qu'un certain délai n'a pas été écoulé. Utilisée pour contrôler la vitesse d'exécution.
- `StepByStepClock`: s'interrompt après la prochaine instruction de dessin et ne continue pas malgré les relances du programme tant que son booléen interne `resumed` n'est pas remis sur `true`. Celui-ci devient à nouveau `false` après l'exécution d'une nouvelle instruction de dessin. Utilisée pour le mode pas à pas
- `TimeoutClock`: s'interrompt lorsqu'un certain délai est dépassé. Reprend à la première relance en attendant à nouveau l'expiration du même délai pour s'interrompre. Utilisée pour empêcher les boucles infinies de prévenir le rafraichissement de la fenêtre JavaFX.
- `AndClock`: fusion de deux `Clock` qui continue si et seulement si les deux horloges l'autorisent. En d'autres termes, s'interrompt dès qu'une des deux horloges le souhaite. Dans notre programme, l'horloge est toujours un `AndClock` entre un `TimeoutClock` et un `PeriodClock` ou `StepByStepClock` selon le mode d'exécution.

3.4. Editeur de code

L'éditeur de code fait avec JavaFX est séparé en deux parties: la vue qui décrit l'interface et le contrôleur qui gère les interactions et fait le lien entre l'interface et les différents éléments cités ci-dessus. Il s'occupe notamment de passer le code source à la chaîne de production et d'utiliser la bonne horloge pour l'exécution selon le mode choisi par l'utilisateur. Il fournit également le contexte graphique du canvas sur lequel dessiner. L'éditeur de code se sert aussi du lexer (analyse lexicale) pour colorer le code tapé par l'utilisateur.

3.5. Gestion des erreurs

Toutes les erreurs liées à un mauvais programme exécuté sont des sous-types de la classe `ChromatynkException`, une exception Java vérifiée. Cela nous permet de facilement gérer le système d'erreurs de notre programme sans nuire à la lisibilité du code avec comme défaut de ne pas pouvoir accumuler les erreurs. Il n'est par exemple pas possible avec ce système d'obtenir plusieurs erreurs de typage si plusieurs parties sont incohérentes vis-à-vis de leurs types. Ce problème peut être résolu par l'utilisation d'une monade (souvent appelée `Validation`) ou d'un effet algébrique. Le premier est

fastidieux à utiliser en Java du fait du manque de sucre syntaxique, de variance et de flexibilité au niveau du typage générique tandis que l'autre n'est tout simplement pas représentable dans le langage.

3.6. Problèmes rencontrés et solutions

3.6.1. Choix du type d'interprétation

Originellement, nous avions choisi d'interpréter directement notre AST via un parcours en profondeur. Cela aurait été plus simple à implémenter et également plus rapide. Cependant, ce type d'interprétation nous empêchait de gérer convenablement plusieurs choses à cause de sa nature récursive:

- Boucles infinies
- Mode pas à pas
- Vitesse du programme

Ces problèmes n'en seraient pas avec un mode d'interprétation itératif. Nous sommes donc partis à la place sur l'interprétation d'une suite d'instructions linéaires (bytecode) par une machine virtuelle, couplée au système d'horloge précédemment décrit.

3.7. Ambiguïté de syntaxe de l'opérateur modulo

Il nous a été demandé par notre professeur d'ajouter l'opérateur modulo. Dans la plupart des langages que nous avons étudiés, cet opérateur est symbolisé par le caractère `%`. Or, ce caractère est déjà utilisé en tant qu'opérateur suffixe (pourcentage) et pose des problèmes d'ambiguïté de syntaxe:

```
5 % 4
```

S'agit-il de l'opération "5 modulo 4" ou de "5%" suivi de "4" (auquel cas il y aurait une erreur de syntaxe)?

Pour palier à ce problème, nous avons choisi d'utiliser le mot-clé `MOD` à la place:

```
5 MOD 4
```

3.7.1. Erreurs de syntaxe récupérables ou non

Originellement, toutes les erreurs de syntaxe étaient considérées comme récupérables. Ainsi lorsqu'il y avait plusieurs choix possibles (notamment via `firstSucceeding`) si le premier parser échouait, le suivant était essayé. Cela menait à des messages peu intuitifs et trop génériques. Or nous savons dans certains cas que si un élément est incorrect alors il s'agit sans aucun doute d'une erreur de syntaxe et non pas d'une mauvaise branche dans notre arbre de possibilités.

```
FOR TO 4
```

Nous savons ici que s'il n'y a pas d'identifiant après le `FOR` alors il s'agit bel et bien d'une erreur: pas besoin de vérifier les autres possibilités.

Nous avons donc introduit le concept d'"erreur fatale". Une erreur fatale est une erreur irrécupérable qui doit simplement être propagée sans être attrapée par un `try/catch`. Ainsi, lorsqu'il n'y a pas d'identifiant après le `FOR` alors l'erreur pertinente est directement levée sans tester le `WHILE`, le `IF`, etc.

4. Manuel d'utilisation

Le projet a été testé sur Windows et Linux. Il devrait également être compatible avec MacOS. Il nécessite d'avoir Java 21 installé sur l'appareil hôte.

4.1. Cloner le projet

Vous pouvez télécharger le zip du projet sur GitHub ou utiliser la commande suivante (nécessite d'avoir Git installé):

```
git clone https://github.com/cytech-ing1-gi22/chromatynk.git
```

Si vous avez Maven installé sur votre machine (généralement embarqué avec Java sous Linux), vous pouvez remplacer toutes les occurrences de `./mvnw` par `mvn` dans les commandes ci-dessous. Ces commandes partent également du principe que l'utilisateur est sous Linux. Sous Windows, utilisez `mvn` ou `./mvnw.cmd`.

4.2. Compiler le projet

Pour compiler le projet, vous devez exécuter à la racine du projet la commande suivante:

```
./mvnw compile
```

4.3. Lancer le projet

Pour lancer le projet, vous devez exécuter à la racine du projet la commande suivante:

```
./mvnw javafx:run
```

4.4. Lancer les tests unitaires

Pour lancer les tests unitaires, vous devez exécuter à la racine du projet la commande suivante:

```
./mvnw test
```

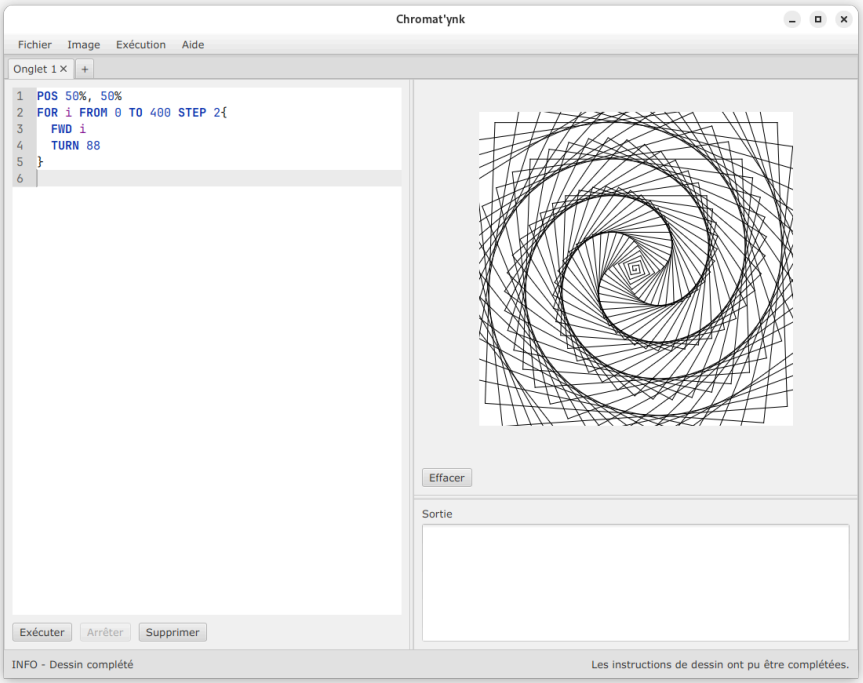
4.5. Vider le cache

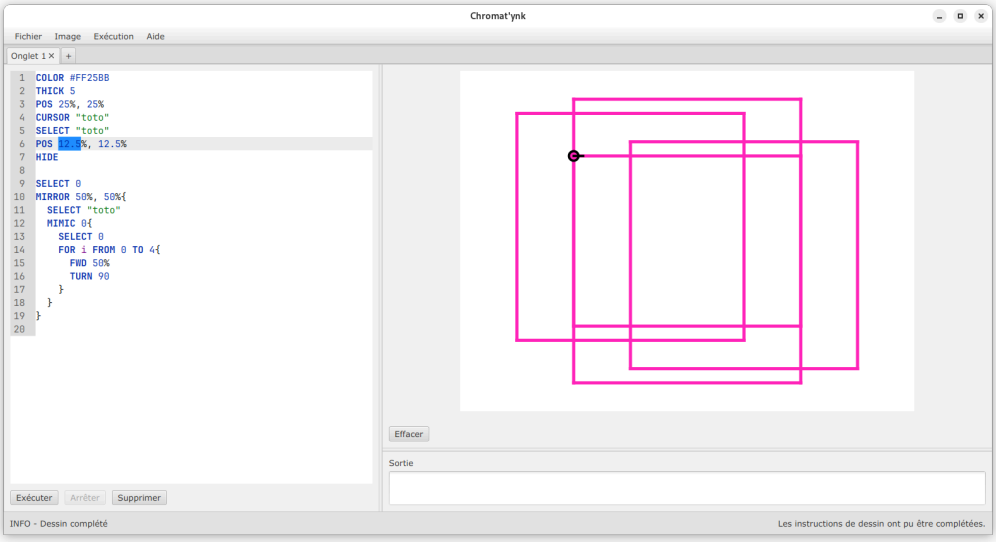
Si le cache de compilation de Maven n'a pas été mis à jour après modification du code ou lorsqu'une erreur probablement liée au cache survient, il convient de le vider en exécutant la commande suivante:

```
./mvnw clean
```

5. Galerie

Editeur de code: fenêtre principale





Editeur de code: fenêtre de configuration des dimensions du canvas

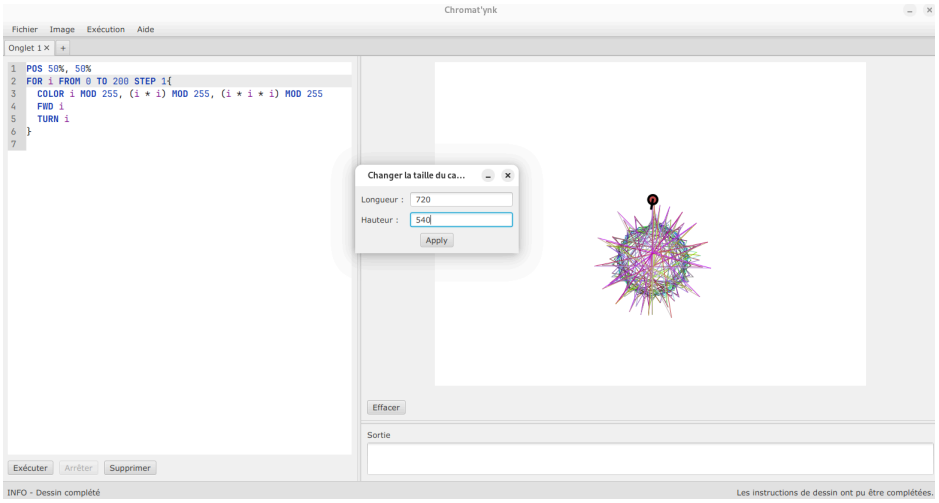


Diagramme de classe du projet

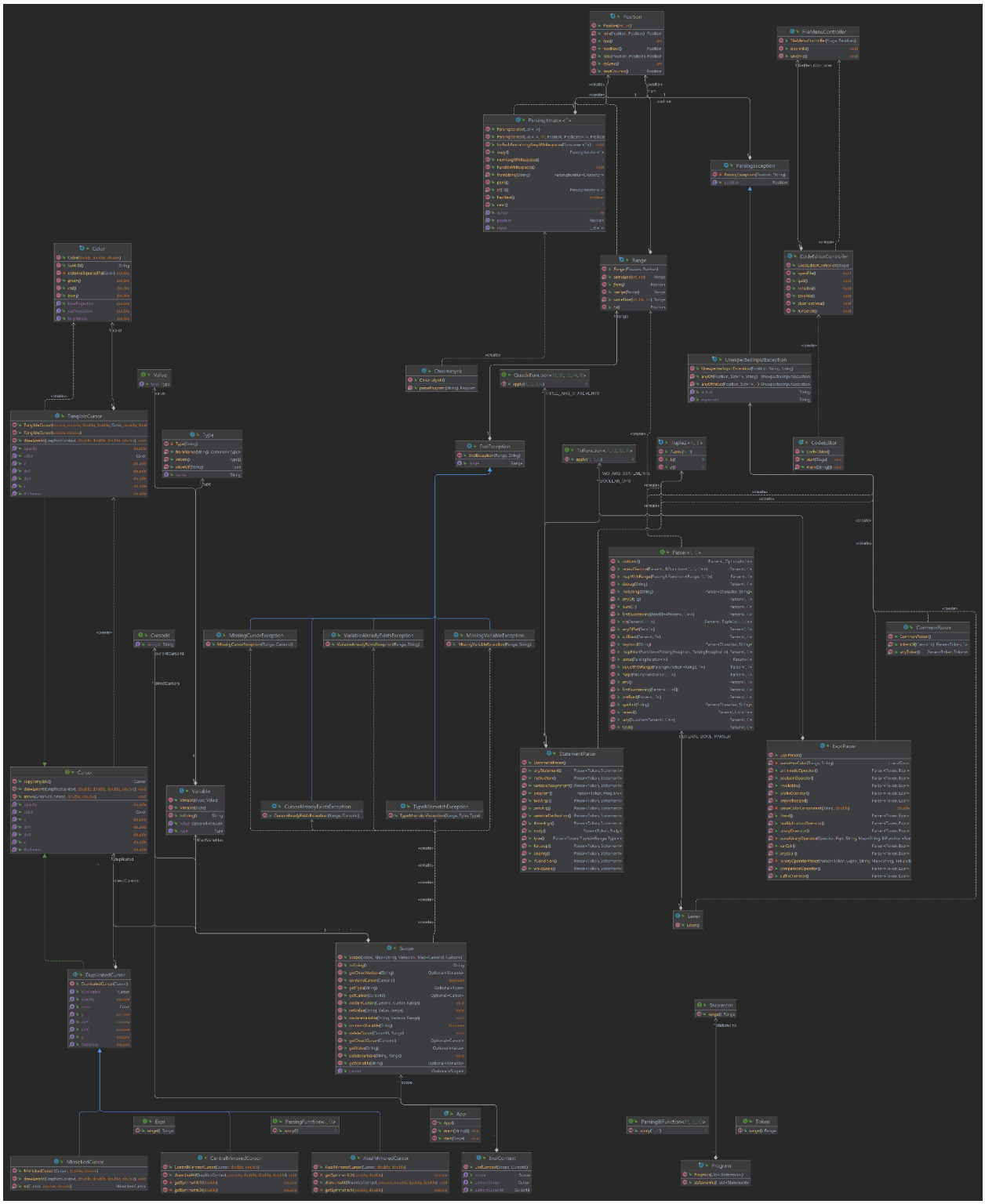


Diagramme de cas d’utilisation