RAPPORT

Equipe :

Ing1 GI 3

- **André Victor**
- **Bogaër Youenn**
- **Piam Lyz**
- **Ion Omkar**
- **Belghiti Mohamed Amin**

Enseignant-tuteur : **M. Khalil Bachiri**

Ce document est le rapport final du projet logiciel d'Ing1 GI du groupe 28.

Sommaire

Sujet :

Le sujet de ce projet est la création d'une application de dessin qui utilise un langage d'instructions. L'utilisateur a en face de lui une zone de dessin et une zone de texte où il peut entrer des instructions (FWD, COLOR, PRESS…) lui permettant de tracer des formes. Il peut aussi télécharger l'image qu'il a réalisée ou importer une série d'instructions déjà complète et constater le résultat.
Nous avons choisi ce sujet car il est selon nous le plus satisfaisant une fois complété, avec des résultats très visuels à travers des dessins et formes géométriques. De plus, il semblait assez complet en termes de nouvelles compétences à apprendre.

Choix de conceptions :

- L'utilisateur doit entrer manuellement ses commandes dans une boîte de texte et l'interpréteur applique la commande ou la rejette.
- Les blocs d'instructions sont délimités par des accolades.
- Les booléens ont comme valeur "true" ou "false".
- Opérateurs logiques AND, OR, NOT.

Organisation et flux de travail :

Outils :

Pour communiquer, nous avons mis en place un serveur discord et le dépôt GitHub dès le début. Le dépôt Git à servi uniquement à stocker l'avancement de l'application. Le serveur discord avait des salons dédiés pour planifier des sessions de travail, partager des ressources, des captures d'écrans et un salon général pour les discussions. La planification, la programmation en groupe, les démonstrations et globalement tout le travail s'est fait sur discord, principalement en vocal.
Pour programmer ensemble, nous hébergions une session Code With Me sur l'un des pc pour pouvoir travailler en même temps sur le programme grâce à Intellij. Avant cela, il était compliqué pour nous de travailler en parallèle à cause du besoin constant de mettre à jour le programme par rapport aux autres.

Flux de travail :

Au démarrage du projet nous avons voulu mettre en place une conception complète et exhaustive pour faciliter l'organisation et la répartition des tâches.
Cela nous a au moins permis d'appréhender l'idée générale de l'application et les attentes quant aux fonctionnalités (bien comprendre l'usage de chaque commande). Cependant nous n'avons pas utilisé à bon escient des outils comme le diagramme de classes, ou des diagrammes de cas d'utilisation et d'activités, qui auraient pu nous faire gagner beaucoup de temps par la suite. La conception incomplète a fait que nous avons dû jeter une partie du travail réalisé par Omkar, car nous avions changé d'avis sur la façon d'implémenter le langage d'instructions (cf. partie Problèmes sur le plan technique).

À la fin de la première semaine, nous n'avions pas commencé à programmer, c'est après notre entretien avec notre tuteur M. Khalil Bachiri que nous avons mis en place une répartition des tâches pour la semaine suivante : Lyz et Victor devaient s'interesser à l'interface graphique GUI en JavaFx, Amin et Omkar à l'interpréteur de commandes et à l'implémentation du langage d'instructions, et Youenn devait continuer la conception et s'intéresser aux exceptions et leur gestions.

La semaine du 13/05 a été peu productive et il était difficile de réunir tout le monde à cause de l'emploi du temps de certains, et nous avions du mal a établir les étapes à suivre pour avancer.

La majorité du travail s'est fait lors de la dernière semaine du rendu.
Les difficultés pour travailler efficacement en groupe, et qui nous ont amenés à prendre autant de retard, ont été diverses : difficultés à programmer en groupe avant d'utiliser les outils d'Intellij, méconnaissance de l'outil GitHub, conception peu concluante qui ne nous a pas aidé à démarrer (exemple premiers diagrammes de classes), manque de disponibilité de plusieurs membres de l'équipe tout au long du mois.
Le retard accumulé a fait que nous n'avons pas pu faire autant de tests et de débogage que nous le voulions.

Problèmes sur le plan technique :

L'interface graphique :
Nous avons décidé de tout de suite s'intéresser à l'interface graphique puisque c'est la version demandée pour la présentation.
Cela signifie que nous n'avons pas utilisé notre version en ligne de commande pour la programmation, elle a été créée uniquement car demandée dans le rendu. Amin a été chargé de créer la version en ligne de commande à partir de la version déjà existante possédant une interface.

Dans un premier temps, nous avons essayé de créer des vues avec la structure suivante :
- une vue principale layout.fxml qui sert de conteneur des autres vues;
- une vue menu.fxml pour le menu d'actions, composé de boutons pour créer un curseur, exécuter des instructions simples ou un fichier au format approprié (.txt), sauvegarder ou supprimer un dessin;
- deux vues terminal.fxml et history.fxml qui correspondent respectivement à la boîte de texte et à l'historique des exécutions (instructions simples, éventuelles erreurs et exécution de fichiers).

Toutefois, nos recherches sur Scene Builder n'ont pas été fructueuses. Nous avons rencontré de nombreux problèmes, les principaux étant pour les fichiers .java de récupérer les ressources des fichiers .fxml (erreurs "Location is not set"). De plus, il s'est avéré impossible d'afficher les vues dans layout.fxml (seul le layout s'affichait) même après avoir fait le point avec notre encadrant, le terminal n'affichant pas de message d'erreur.
Ainsi, nous avons opté pour une approche plus simple.

Notre MVC (Modèle-Vue-Controller) est constitué d'une unique vue qui a pour contrôleur l'application. La vue possède les mêmes éléments graphiques que ceux utilisés avant. L'interface possède également un curseur pour choisir le temps d'attente entre deux instructions (de 1 ms à 2 s) et un tableau pour choisir la couleur du fond de l'image. Les images sont gérées avec une librairie javafx-swing.

Gestion des blocs :

Pour les blocs nous avons décidé de les délimiter grâce aux crochets comme en java, si il n'y en a pas alors on a une erreur qui s'affiche dans l'historique. Cette délimitation nous empêche d'écrire le contenu d'un bloc sur différentes lignes mais cela nous permet de nous assurer que tout le bloc soit traité en même temps.

Bloc IF :

Pour gérer les expressions booléennes nous avons créé la méthode *Interpreter.evaluateBooleanExpression(String expression)* qui recherche tous les éléments qui composent une expression booléenne. Notre méthode ne gère pas les parenthèses et c'est quelque chose que nous avons remarqué tardivement donc nous avons décidé de ne pas le faire. Cela signifie que les expressions du type : *true AND (x == 1 AND (y ==2))*, ne peuvent pas être implementées. Seulement les suites d'expressions simples délimitées par des AND et des OR.

Bloc FOR :

La méthode qui gère les boucles FOR réécrit une instruction avec le bon nombre d'itérations qui est renvoyé dans la méthode interpret. On peut choisir le début et le pas mais si on ne rentre pas step et from les valeurs par défauts sont 0 pour le début et 1 pour le pas. Nous vérifions que notre boucle ne soit pas trop grande pour ne pas faire planter le programme.

Bloc MIRROR:

- Symétrie centrale

On crée un nouveau curseur temporaire en faisant une symétrie centrale et en créant un curseur temporaire puis on appelle executeWithTmpCursor pour que les 2 s'éxécute en même temps.

- Symétrie axiale

Dans le cas où le curseur n'est pas sur le segment formé par les deux points (auquel cas le curseur temporaire a les mêmes coordonnées), le but est d'abord de trouver son projeté orthogonal sur la droite puis de faire une symétrie axiale par rapport à ce-dernier.
Nous avons réussi à mettre en équation cette approche mais nous n'avons pas réussi à implémenter la solution.

$$\begin{bmatrix} p_1.x - p_0.x & p_1.y - p_0.y \\ p_0.y - p_1.y & p_1.x - p_0.x \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix} = -\begin{bmatrix} -q.x(p_1.x - p_0.x) - q.y(p_1.y - p_0.y) \\ -p_0.y(p_1.x - p_0.x) + p_0.x(p_1.y - p_0.y) \end{bmatrix}$$

q étant les coordonnées du curseur, p0 et p1 celles des deux points, X et Y les coordonnées du projeté orthogonal.

Le délai entre les instructions :

Le délai entre les instructions se configure par une barre glissante avec plusieurs niveaux dans l'interface. La gestion du délai s'est fait dans la boucle permettant d'identifier les différentes commandes. Cependant, l'application attendait que toutes les instructions soient terminées pour afficher le résultat.

Il est impossible de faire un FOR dans un bloc qui contient un curseur temporaire (MIMIC/MIRROR) car pour régler un problème rencontré, concernant l'instruction SELECT, nous avons dû créer une variable de classe temporaire usedCursor or lors de l'exécution d'un FOR, cette variable temporaire est remis à jour ce qui fait que notre la variable temporaire n'effectue pas les actions demandé dans le FOR.

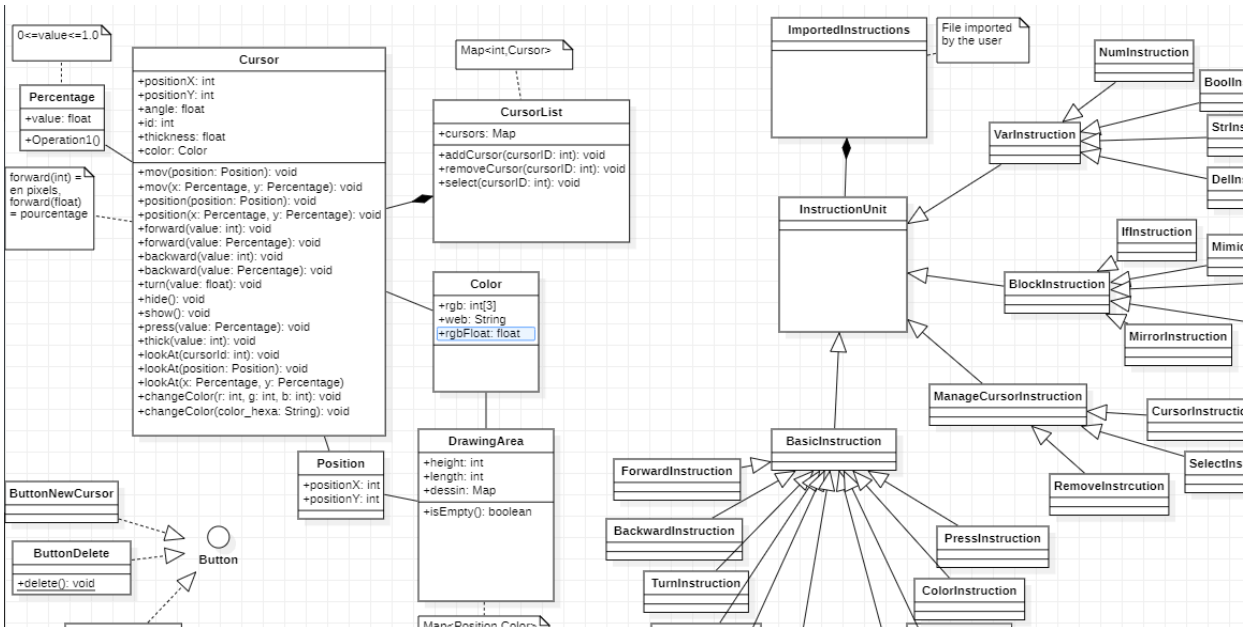
Pour tout les blocs avec un curseur temporaire une des difficultés majeurs était l'exécution avec du délai, pour y remédier nous avons dû créer une méthode qui contient aussi une Timeline pour que les 2 exécution puissent se faire mais surtout pour qu'après la pause le curseur temporaire soit supprimé car la timeLine ne met pas en pause tous les threads donc le curseur temporaire était supprimé avant que les instructions qui le concernent soient finis.

<u>Le langage d'instructions :</u>

L'implémentation du langage d'instructions a été une problématique que nous avons tout de suite identifié. Comment reconnaître les symboles et exécuter le bon comportement en fonction de cela ?
Pour le langage nous voulions d'abord que, d'une certaine façon, on puisse stocker toute la grammaire dans une classe et que le bon comportement soit associé à chaque symbole. C'est ce qu'avait commencé à faire Omkar et que nous avons dû abandonner par la suite. Il n'en reste rien aujourd'hui dans le code car nous avons décidé de stocker tous le vocabulaire et les méthodes associées à chaque instructions dans une seule classe *Interpreter*. Dans *Interpreter,* l'instruction en entrée passe dans un switch qui identifie la commande voulue et exécute la bonne méthode (voir diagramme de classe final).

Lors de la conception du premier diagramme de classe, nous hésitions entre une classe *Interpreter* qui contient toutes les instructions possibles et leurs méthodes ou une classe associée à chaque instructions. Nous avons décidé qu'une seule classe serait la façon la plus simple de faire, étant donné le nombre de classes à créer.

Premier diagramme de classe (version avec une classe par instruction) :



Les curseurs et leurs attributs :

Curseurs :
Les curseurs sont les objets les plus importants pour cette application. La classe Cursor permet d'instancier des curseurs avec leurs attributs (position, couleur, épaisseur, opacité…) ainsi que de stocker des méthodes pour les modifier leur état.
Pour la gestion des curseurs actifs nous avons décidé d'utiliser une Map car cela semblait être le plus simple, un curseur correspond à une clé (l'attribut id de l'objet Cursor).
Les curseurs sont affichés sur une interface différente du dessin car ils sont éphémères (nous gérons notamment des curseurs temporaires dans les blocs MIMIC et MIRROR), et qu'ils ne doivent pas apparaître sur le dessin final (quand l'utilisateur télécharge l'image).

Angles :
Nous pensions avoir correctement implémenté la gestion des angles jusqu'à très tard dans le développement mais lors des tests, notamment de la commande LOOKAT (qui permet de diriger le curseur dans une direction souhaitée), nous nous sommes rendu compte que les angles étaient incorrects. La raison vient du fait que le modulo % (remainder) en Java ne fonctionne ni avec des réels (double) ni avec des négatifs. Une fois que nous avons détecté cela, ça a été réglé rapidement. La trace de ce travail se remarque car les angles dans le programme sont des floats, et non des doubles comme initialement.

Couleurs :
La façon dont sont utilisées les couleurs dans le cahier des charges implique que nous devions gérer trois format différents : RGB (en entiers), RGB (en réels de 0 à 1) et Web (type #RRGGBB en hexadécimal). Il nous a paru évident qu'une classe Color devait être créée. Par la suite nous avons découvert qu'une classe Color existait déjà en Java, mais nous avons décidé de conserver notre classe personnalisée (renommée Colorj) car elle s'adapte parfaitement à nos besoins pour la couleur des curseurs. A noter que la classe Color de Java est tout de même utilisée pour l'interface graphique.

Autres :

Gestion de la pause :
Pour pouvoir implémenter le délai entre chaque instructions,nous avons voulu utiliser la méthode Thread.sleep mais ça mettait en pause tous les threads dont celui de JavaFx ce qui empêchait la mise à jour du dessin au fur et à mesure. Plusieurs autres méthodes ont

été explorées, au final nous avons utilisé la classe TimeLine, Duration et KeyFrame qui permettent de mettre le Thread actuel en pause.

Variables :
Au vu du cahier des charges, nous les avons représentées avec une classe abstraite Variable. Elle possède trois classes filles Num, Str et Bool qui correspondent aux instructions d'initialisation. Les variables sont identifiées par leur nom. La méthode statique *Variable.isValidName*(*String name*) assure que les noms commencent uniquement par des lettres en s'appuyant sur des regex.
Les variables numériques peuvent être utilisées dans des instructions prenant un seul argument. Pour les distinguer de simples valeurs numériques, la méthode statique *Num.isName*(*String name*) détermine si le nom d'une variable numérique est bien utilisé. Bien que nous n'ayons pas eu le temps de traiter les instructions ayant deux arguments, l'approche reste similaire en vérifiant si les deux, l'un des deux ou aucun n'est une variable.

Exceptions :
Dans un premier temps, la plupart des exceptions correspondaient à une mauvaise saisie des instructions alors nous utilisions *IllegalArgumentException*. Mais pour clarifier le code et faciliter le débogage nous avons décidé de créer les exceptions personnalisées suivantes :
- UnknownCommandException qui gère les éventuelles erreurs de syntaxe dans les instructions exécutées;
- OutOfPositionException qui soulève une exception pendant l'exécution d'une instruction qui déplacerait le curseur sélectionné hors de la zone de dessin;
- UndefinedVariableException qui s'assure que les variables utilisées dans les instructions ont été définies au préalable;
- FileException qui gère l'exécution d'un fichier importé;
- LoopSyntaxException qui gère la syntaxe à l'intérieur des boucles;
- NewCursorException qui gère les problèmes à la création de nouveaux curseurs;
- MirrorException et MimicException.

Annexe :

Diagramme de cas d'utilisation :

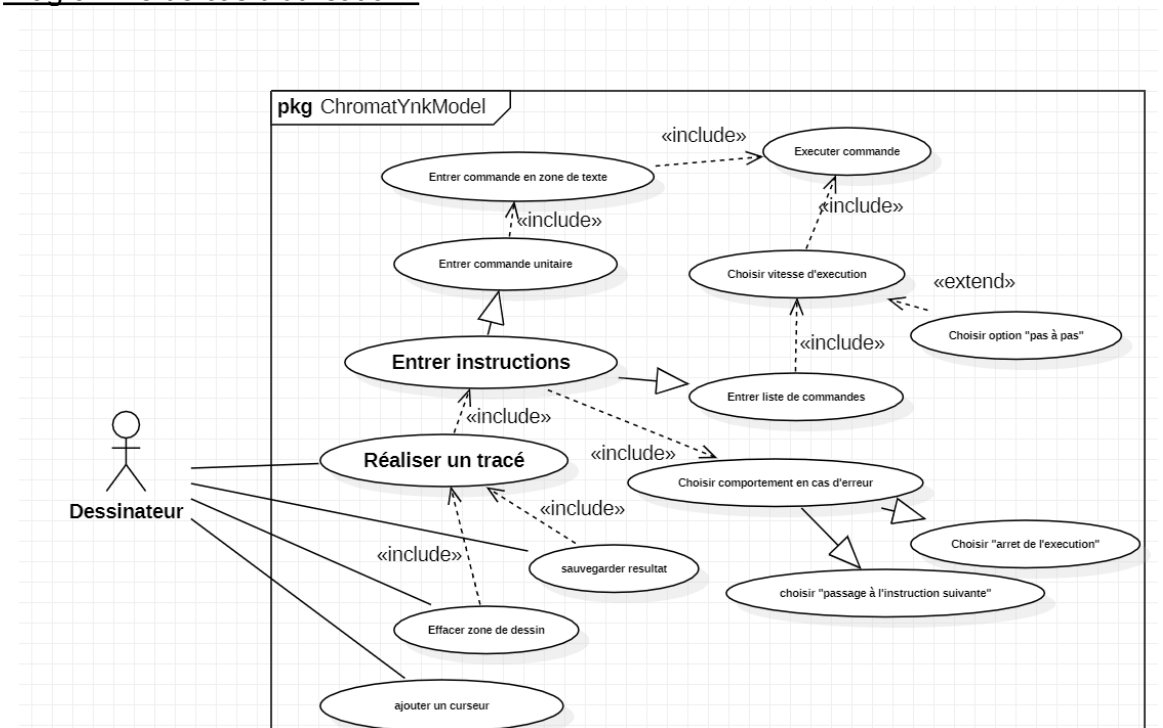


Diagramme de Classes final :