Tom LACHENAUD ZIMMERMANN
Quentin JEAN
Nawel ZEBOUDJ
Maxime PAUCHON
Assia BROUZIYNE

RAPPORT PROJET GÉNIE LOGICIEL

CHROMAT'YNK

Sommaire :

1. Introduction

Pour ce projet de Génie Logiciel, nous avons eu le choix entre 4 différents sujets : Chromat'Ynk, CodYngame, CY-Books et Généalogie. Après s'être mis en groupe de 5, nous avons étudié chaque projet et leur objectif final afin de choisir celui qui nous correspondait le mieux et qui nous semblait être le plus stimulant à faire. Nous avons donc décidé de fixer notre choix sur le premier sujet : Chromat'Ynk.

Chromat'Ynk qu'est-ce que c'est ? C'est un projet ayant pour but de créer une application permettant à l'utilisateur de dessiner des formes quelconque, a partir de lignes, en utilisant des instructions simples comme : avancer, reculer, tourner, etc. Cependant, la difficulté principale de ce projet était de créer et d'interpréter un langage d'instructions permettant de dessiner ces formes quelconque.

Le but principal de ce projet étant de créer une application en Java et JavaFX en trois semaines, nous avons donc dû nous organiser au sein de l'équipe afin de rendre un programme fonctionnel et complet, répondant au cahier des charges qui nous a été fourni. Au cours de ce rapport nous aborderons dans un premier temps comment nous nous sommes organisés en tant qu'équipe pour travailler le plus efficacement possible. Et dans un second temps nous verrons comment gérer les éventuels problèmes rencontrés et comment nous avons réussi à les solutionner.

2. Gestion de projet

Tout au long du mois de mai, dans le cadre de l'UE Projet, nous avons eu un projet Génie Logiciel à réaliser sachant que les trois premières semaines ont été dédiées au développement du projet et la dernière semaine à la préparation de notre soutenance.

Tout d'abord, afin de réaliser au mieux ce projet, nous avons dû nous organiser de façon à ce que nous ne perdions pas de temps, considérant le délai qui nous a été imposé. Pour ce faire, nous avons créé trois groupes :

- Un groupe s'occupant de toute la partie Java du projet, rassemblant toutes les fonctionnalités de base du projet. Ce groupe s'est composé de Nawel et Quentin.
- Un groupe s'occupant de toute la partie JavaFX du projet. Étant donné que notre projet vise à dessiner des formes sur une fenêtre de dessin, notre équipe a récupéré toutes les méthodes de base du projet en Java pour faire en sorte que l'affichage corresponde aux attentes. Ce groupe s'est composé de Maxime et Tom.
- Assia qui s'est occupée, seule, de gérer toutes les erreurs nécessaires au bon déroulement du projet.

Une fois que nous avons décidé de cette organisation, nous avons commencé par construire une première version du diagramme de classe afin de se rendre compte de la construction générale du projet. Ceci étant fait, nous avons réparti les tâches dans les différents groupes et avons commencé par créer un dépôt de code GitHub afin que chacun dépose son code au fur et à mesure qu'il code pour garantir un avancement optimal et sans duplication. Afin de mieux s'organiser, nous avons décidé de créer des branches différentes pour mieux découper les parties et pour que chacun puisse tester des fonctionnalités avant de le mettre sur la branche principale, sans rendre le code obsolète.

Ensuite, il est important de noter que notre projet s'est fondé sur une chose : la communication. En effet, pour garantir le bon avancement du projet et pour ne pas prendre de retard, nous fixions des réunions d'équipes tous les trois jours. En plus de ces réunions, nous nous tenions informés de nos problèmes et solutions ainsi que de nos avancés via un groupe télégramme et Discord. Aussi, étant donné la construction de groupes à l'intérieur de notre équipe, il faut préciser que nous faisions également des réunions en distanciel ou présentiel entre groupes en plus des réunions d'équipe afin que nous puissions chacun avancé de notre côté.

3. Analyse de projet

Étant donné que nous avons découpé le projet en plusieurs parties, il est important d'analyser le projet par partie.

Tout d'abord pour la partie Java il nous a fallu créer les classes qui vont servir de support pour les instructions et le JavaFX tel que les classes associé aux points, aux positions, aux couleurs ainsi que le CursorManager. Les seules difficultés rencontrées lors de cette partie étaient de penser et de rassembler les attributs et les méthodes nécessaires dans chaque classe dont nous allions avoir besoin pour la création des instructions. Ce qui a mené à la création de nouvelles méthodes, la suppression de certaines et l'ajout d'attributs tout au long du projet dans les premières classes que nous avons créé.

Une fois ces bases posées nous avons commencé par la création des simples instructions qui sont les instructions les plus simple qui permettent de donner les informations nécessaires au javaFX pour afficher le résultat des instructions, ce qui passent par l'actualisation des données du curseur utilisé et la gestion des curseurs créé par l'utilisateur. Pour cette partie nous n'avons pas vraiment eu de problème nous avons juste relevé d'autres possibilités d'exploitation de la classe instruction. Nous avons utilisé un attribut nommé type sur lequel on faisait un switch pour exécuter la simple instruction correspondante mais nous aurions pu potentiellement passer par des sous classes héritant de simple instruction pour chacune des instructions.

Une fois ces simples instructions fonctionnelles nous sommes passés aux blocs d'instructions. Pour les blocs d'instructions nous avions anticipé l'utilité que simple instruction et blocs instructions soit tous les deux des sous classes d'une même classe instruction qui permettra de faciliter l'imbrication de plusieurs blocs d'instructions. Donc avant que nous créions les variables les premiers blocs ne nous ont pas posé de problème particulier. Les seuls blocs nous ayant posé problèmes étaient MIMIC et MIRROR. Pour MIMIC le problème était principalement la compréhension de son fonctionnement selon la consigne qui n'était pas très claire. On a donc au début créé plusieurs fonctions mais finalement nous avons eu les retours sur la fonction par le créateur du sujet, ce qui nous a permis de faire une version finale équivalente à la demande de base du sujet. Pour MIRROR il y eu deux problèmes principaux que nous avons rencontré et qui nous ont pris beaucoup de temps. Le premier problème est directement dans la logique de la méthode, exécuter les

instructions simples en miroir cela ne posait pas de problème mais lorsque celle-ci se trouvait dans les blocs d'instructions cela devenait problématique. On a donc créé une méthode mirror instruction pour chaque instructions. Ensuite le deuxième problème était l'imbrication de deux MIRROR pour avoir une symétrie sur chaque point et correspondant à la bonne instruction miroir issues du bon curseur. Pour résoudre ce problème il a fallu penser à l'inversement que cela créait.

Nous avons ensuite ajouté la possibilité de créer des variables grâce à une nouvelle classe d'instructions qui est dédiée à cet usage. Les variables sont rangées dans une map afin de s'assurer d'avoir accès à toutes les valeurs depuis leur clé. Elles peuvent être de 3 types: numérique, booléen ou chaine de caractère. La subtilité pour la création de ces variables à été de gérer les cas où les numériques sont entiers ou décimales en passant par la classe abstraites Number de laquelle hérite Integer et Double. Une fois ce problème de type réglé, il a fallu revoir le traitement des paramètres des instructions pour qu'ils puissent être sous forme de variable. Le passage de chaine de caractère à valeur numérique associé à une variable ou non a été long à gérer. Une méthode est finalement dédiée à cet usage dans chaque classe d'instruction.

La transformation de chaine de caractère en condition a également été longue et difficile à gérer. La gestion simultanée d'opérateur de comparaison, d'opérateur logique et de variable potentiel a amené à un grand nombre de tests et de tentatives avant d'arriver aux méthodes fonctionnelles présentes dans la classe de blocs d'instructions.

Enfin, la dernière partie travaillée sur le plan Java a été l'analyseur de texte pour transformer le texte de l'utilisateur en instruction pouvant être des simples instructions, instructions de variable ou blocs d'instructions. Le choix ayant répondu à toutes les contraintes imposées par les différentes classes d'instruction a été d'établir une liste de toutes les instructions à exécuter en utilisant des délimitateurs de texte (';','{','{') et des traducteurs de chaque type d'instructions. Le problème rencontré à ce moment a alors été la possibilité de blocs imbriqués. Construire une pile pour gérer la construction de chaque bloc a été notre solution à ce problème. Les erreurs syntaxiques n'ont pas pu toutes être implémentées, nous avons toutefois fait en sorte que le programme ne s'arrête pas subitement dans le cas d'une de ses quelques erreurs syntaxiques non-gérées même si la cause n'est pas forcément expliquée.

Ensuite, pour la partie JavaFX, il nous a été nécessaire de trouver comment faire pour afficher une ligne ainsi qu'un curseur. Pour toutes les instructions en rapport avec les traits cela a été assez simple puisque qu'une classe en JavaFX existe déjà avec toutes les méthodes

pour modifier l'opacité, la couleur etc. Cependant, la partie la plus complexe a été la gestion de curseurs. En effet, une classe Cursor existe déjà en JavaFX mais elle ne correspond pas complètement à ce que l'on veut, il nous a donc fallu créer une classe avec toutes les méthodes d'affichage nécessaire. Au début nous pensions qu'il fallait que le curseur dessine le trait, chose qui nous a semblé impossible à faire, du moins avec la méthode que nous utilisons pour tracer les lignes. Pour résoudre cela, nous avons décidé de récupérer le point d'arrivée lorsque l'on crée une ligne et de positionner les coordonnées du curseur sur celle du point d'arrivée.
Une fois le problème de curseur réglé, le deuxième problème que nous avons rencontré est celui de la gestion de plusieurs curseurs sur une même fenêtre. En effet, en termes de code Java pure, nous avons créé une liste de curseurs qui supprime, ajoute et sélectionne le curseur que l'on souhaite mais au moment de la gestion JavaFX cela a été plus complexe. Pour ajouter un curseur, il suffit simplement de l'ajouter à la root de la scène mais lorsque l'on voulait supprimer un curseur ou sélectionner un curseur en particulier, cela supprimait tout le contenu de la scène ou ne sélectionnait pas de curseur. De plus, on ne pouvait pas simplement sélectionner un élément de la scène car chaque élément est de type Node alors que nos curseurs ne le sont pas, nous ne pouvions donc pas les récupérer par ID. Pour cela, les curseurs que nous avons ajoutés à la scène sont une représentation d'un triangle de type Polygon ayant pour paramètre tous ceux du curseur. L'avantage de faire cela est que nous avons pu initialisé un ID récupérable dans la liste d'éléments de la scène ce qui nous a permis de parcourir l'ensemble des éléments de la scène afin de sélectionner/supprimer le curseur que l'on souhaite.
Enfin, concernant la partie d'instructions venant d'un fichier texte, il nous avait été demandé dans le cahier des charges de réaliser les instructions pas à pas ou selon une vitesse d'instruction. Malheureusement, après de nombreux essais nous n'avons pas réussi à implémenter cette fonctionnalité. En effet, nous nous sommes renseigné sur la documentation de Java et de JavaFX sur des classes comme ScheduledExecutorService et TimeUnit qui, testé sur des programmes simples fonctionnent mais lorsque appliqué sur le programme principale n'affichent qu'une partie des instructions.

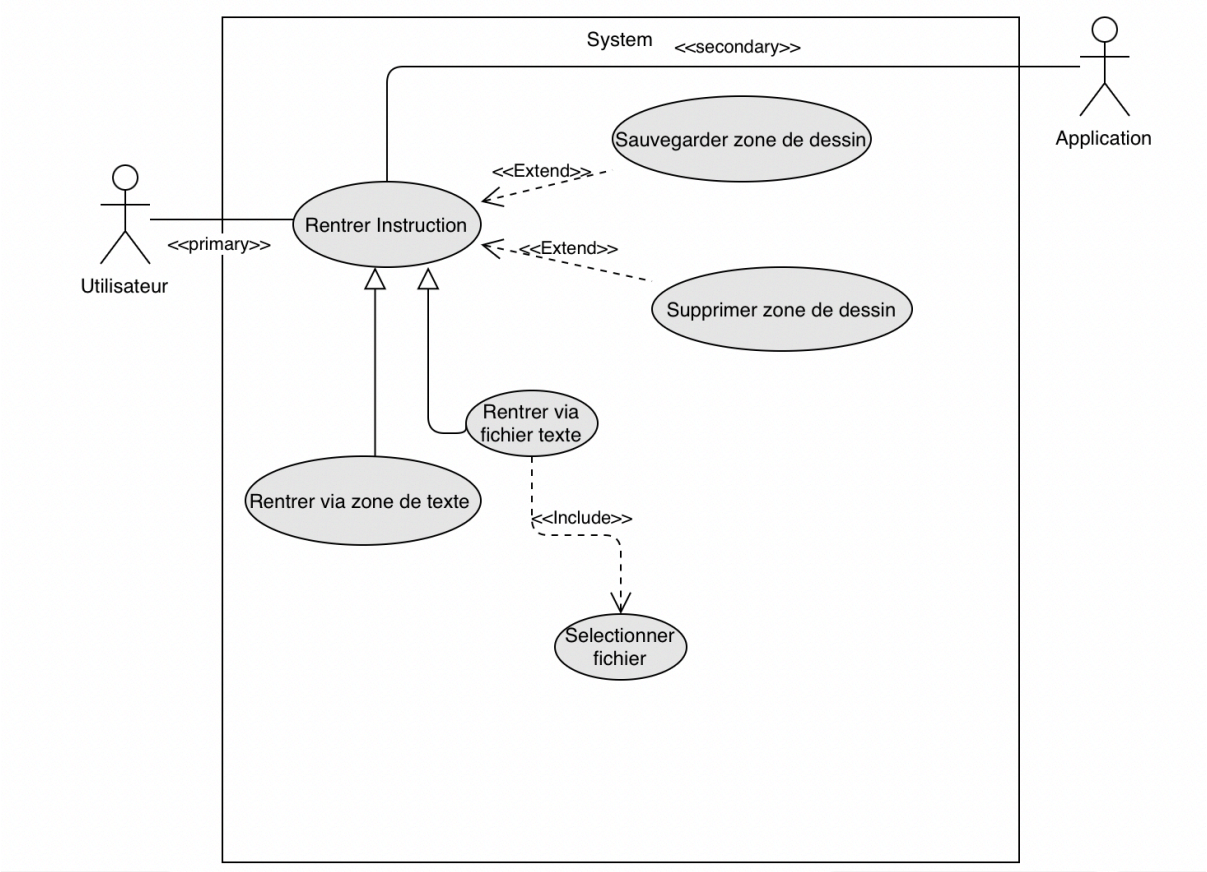
Pour finir, les erreurs ont dû être fixées. En effet, au fur et à mesure du développement de notre application, des exceptions ont été utilisées telles que "IllegalArgumentException"ou "NumberFormatException". Ces exceptions nous ont servi de repère car elles mettaient en évidence les conditions de fonctionnement des différentes

méthodes. Cela nous a permis d'y voir plus clair et de reconnaître quel type d'exceptions était attendu et adapté aux différentes situations. En effet, selon les conditions que l'on souhaite respecter, les messages d'erreur diffèrent. L'adaptation des exceptions en fonction de notre classe d'erreur ErrorLogger, qui est une sous-classe de la classe Exception, a été une tâche à effectuer avec soins car des soucis de définition ou d'appels pouvaient rendre les erreurs non fonctionnelles et perturber le reste du code.

Nous avons utilisé différentes techniques telles que "try-catch", ou bien la création d'une instance de la classe ErrorLogger. Nous avons également fait appel à la classe en ajoutant dans la signature de certaines méthodes "throws ErrorLogger". Ainsi, selon la situation, nous avons géré les erreurs de manière locale ou bien de manière globale en utilisant la structure de la classe ErrorLogger. Avec ces différents procédés et techniques, il était parfois compliqué de choisir la combine adaptée au cas, mais nous avons fini par prendre nos repères et y voir plus clair. Sachant que la classe ErrorLogger hérite de la classe Exception qui elle-même hérite de la classe Throwable, ErrorLogger a pu récupérer des méthodes comme "getCause" et "getMessage".

Comme nous ne voulions pas que le programme s'arrête en cas d'erreur détectée, il a été impératif d'utiliser ErrorLogger afin de permettre au programme de continuer son parcours tout en renvoyant l'erreur. De plus, la mise en place d'un gestionnaire global d'exception dans la classe Main a été très utile. Il nous a permis de masquer la pile d'appel de JavaFX qui était affichée en cas d'erreur, ce qui nous empêchait de repérer l'erreur au sein des nombreux messages affichés.

4. Diagramme de cas d’utilisation



5. Diagramme de classes