UTBM : Anthony Dury - Kadir Ercin - Romain Henry

Bases fondamentales de la programmation orientée objet - Pr. GECHTER

Compte-rendu

utbm
université de technologie
Belfort-Montbéliard

Table des matières

Introduction

L'exercice a pour but de nous initier aux principaux concepts de la programmation orientée objet. Plus particulièrement de comprendre et de savoir mettre en oeuvre Java, ainsi que d'acquérir une méthode d'analyse retranscrite dans des diagrammes UML.

Il s'agissait ici d'implémenter soit un jeu de plateau, soit le nouveau jeu de steam "Mini Metro". Nous avons choisi d'adapter le jeu Mini Metro qui nous paraissait plus "user friendly", une adaptation qui ne sera donc pas personnalisé afin de préserver le gameplay.

Le principe de ce jeu est donc de construire un réseau de métro au fur et à mesure de la demande. Il faut donc dessiner des lignes entre les stations afin que les voyageurs puissent se rendre à leur destination. Mais attention, la ville se développe et de plus en plus de stations et d'utilisateurs apparaîtront à des positions aléatoires. La partie prend fin lorsqu'une station atteint son quota maximum de voyageur. De ce fait, nous pouvons et devons repenser notre réseau à chaque instant afin de maximiser son efficacité. Au fil du temps, des train, des lignes, des ponts vous sont offerts pour vous aider. L'objectif est donc de répondre à un maximum de demandes.

Nous avons donc pris le jeu en main avant de procéder à son analyse en reprenant la norme UML. La première partie de ce document fait donc l'objet de description de cette analyse. Puis ensuite de sa conception.

I. Cahier des charges

A. Choix des technologies

Dans un premier temps nous avons donc réfléchis sur l'organisation et la sémantique du projet. De ce fait nous avons créé un dépôt privé sur GitHub afin de faciliter le travail de groupe et de suivre aisément l'évolution du projet, point essentiel sûr pour mener à bien tout projet de développement.

L'implémentation de l'application sera réalisée en JAVA sous l'IDE IntelliJ IDEA.

Nous avons décidé d'utiliser la bibliothèque JavaFX pour gérer l'interface graphique utilisateur afin de découvrir et d'apprendre une autre bibliothèque différente de Swing qui devient obsolète.

Concernant la modélisation UML, nous avons réalisé les diagrammes avec le logiciel StarUML de MKLab.

B. Les fonctionnalités

Les actions possibles de l'utilisateur peuvent être résumés par le diagramme de cas d'utilisation qui suit.

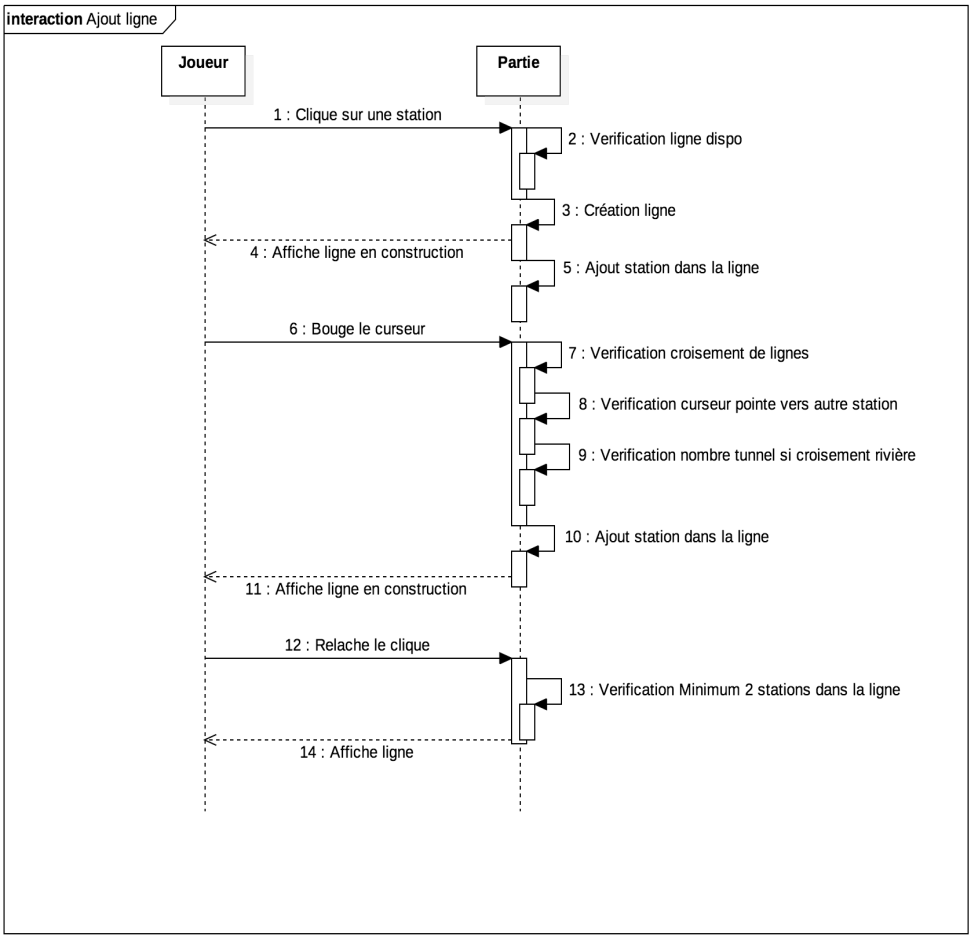
Nous nous concentrerons dans un premier temps sur l'implémentation des mécaniques de bases :

- Gestion des retraits et ajouts des lignes et des stations.

- Déplacements des clients et des trains.
- Apparition des clients et des stations.
- Utilisation d'une carte fixe et unique avec rivière impliquant la gestion des ponts.
- Gestion du temps.

Nous avons pris le choix de ne spécifier uniquement le fonctionnement des utilisations les plus complexes, à l'aide de diagrammes de séquence qui suivent.

- Ajouter une ligne :

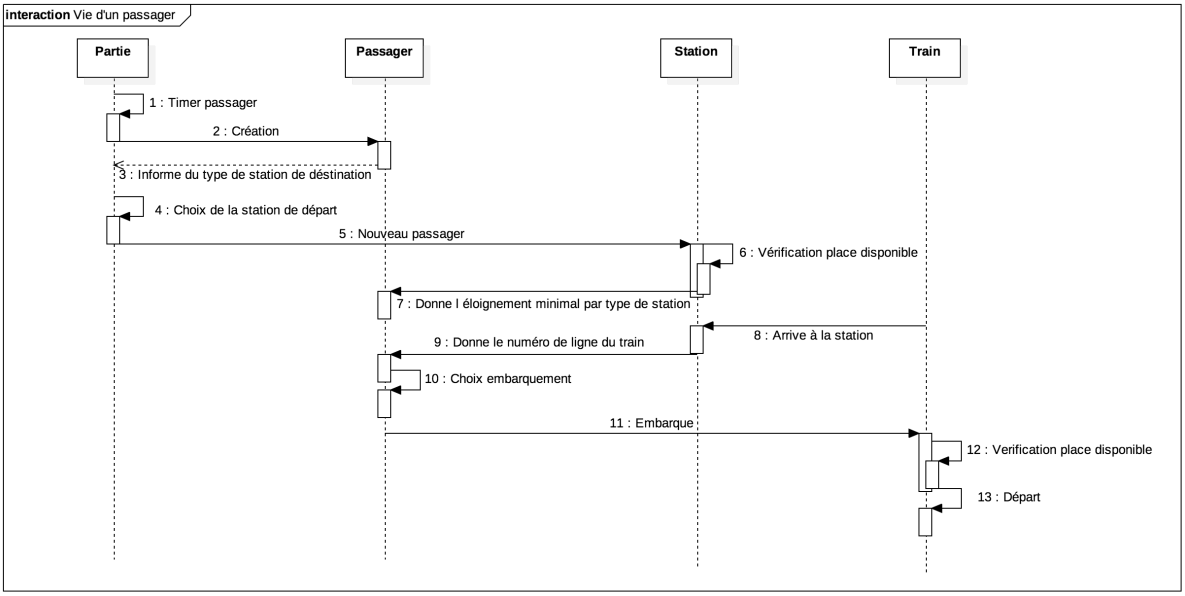


Nous allons utiliser le mécanisme de gestion des exceptions de JAVA afin d'identifier et traiter les erreurs.

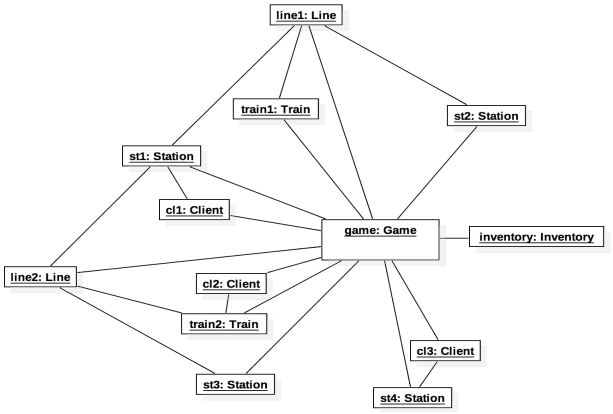
En fonction de notre avancement, voici une liste non exhaustive des fonctionnalités envisagées:

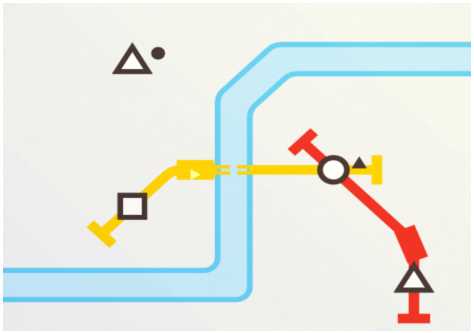
- Pouvoir jouer sur plusieurs cartes.
- Gérer le dézoom sur la carte lorsque le nombre de stations est élevé.
- Sauvegarder/charger une partie.
- Ajouter des fonctionnalités non existantes …

Afin de comprendre l'interaction entre les élément de notre système nous avons produit un diagramme dynamique permettant de mieux comprendre la “vie d’un passager”



Afin d’exprimer un contexte d'exécution et d’avoir une idée des instances des classes à un moment T, nous avons produit un diagramme d’objet représentant le jeu avec différents cas voir ci-dessous.





- la ligne rouge *(line1)* contient une station “triangle” vide *(st2)*, une station “rond” *(st1)* avec un client “triangle” *(cl1)* et un train vide *(train1)*
- la ligne jaune *(line2)* contient une station “carré“ vide *(st3)*, une station “rond”*(st1)* avec un client “triangle” *(cl1)* et un train *(train2)* avec un client “triangle” *(cl2)*
- une station “triangle” contient un client “rond”

II. Organisation du projet

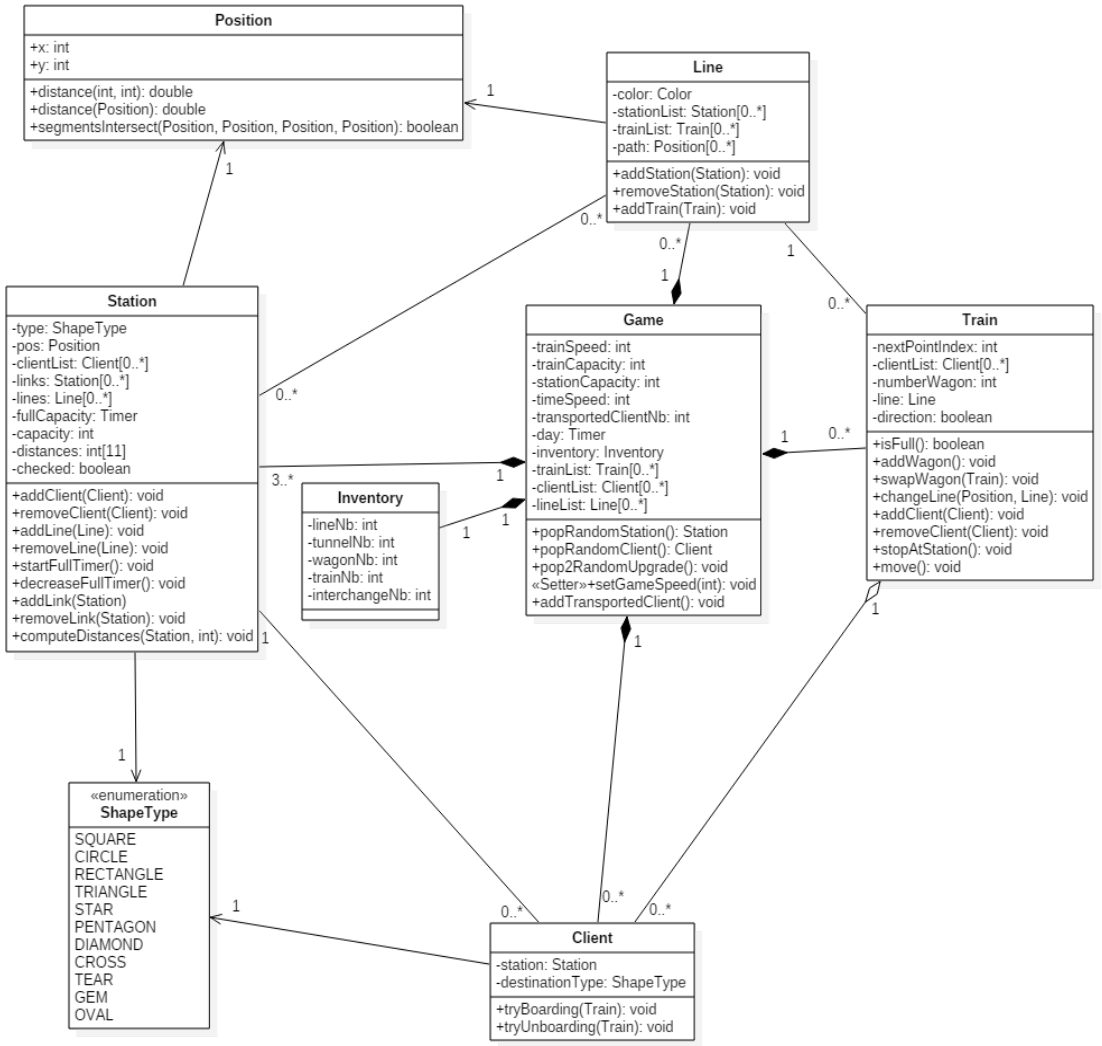
Nous avons décidé de créer des classes pour représenter les clients, les lignes, les trains, les stations.

Nous avons créé la classe Game qui permet de gérer le jeu, c'est à dire que cette classe contiendra les listes des stations, clients, lignes, trains et fera apparaître les clients, les stations et contiendra l'inventaire du joueur, qui est matérialisé par une classe.

Pour gérer le type des formes des clients et stations nous avons créé une énumération ShapeType.

Nous avons rajouté la classe position qui permettra de stocker des coordonnés entières (x, y) et qui gèrera tout ce qui concerne les intersections entre des droites, les distances etc …

Voici le diagramme de classe correspondant :



IV. Implémentation

Différences Implémentation <-> UML :

Nous avions fait la conception UML de notre modèle pour le projet, celle-ci a été en grande partie respectée. En effet, nous avons uniquement rajouté la classe Clock, qui représente l'horloge de MiniMetro et permet d'effectuer l'écoulement du temps, et des jours.

Nous avons ensuite décidé d'essayer de respecter le pattern MVC (model view controller) avec l'utilisation de la librairie graphique javafx.

Pattern MVC

Pour la vue nous avons presque dupliqués toutes les classes du modèles en ajoutant le suffixe fx aux noms des classes : par exemple ,nous avons dans le modèle la classe Station, et dans la vue la classe fxStation, classe Train à fxTrain.

Nous avons donc aussi dupliqués la classe Game, nommé dans la vue GameView. La classe GameView contient des HashMap permettant d'associer un objet du modèle à son équivalent dans la vue. Cela nous permet donc de notifier la vue à partir du modèle, par exemple, lorsque l'on fait apparaître une station dans le modèle, on notifie la vue avec la référence de la station, la vue crée donc l'équivalent dans la vue et les associer avec des HashMap.

Le contrôleur lui permet de mettre à jour le modèle, comme dans les cas suivants : lorsque l'on fait des liens entre stations, le contrôleur mets à jour le modèle.

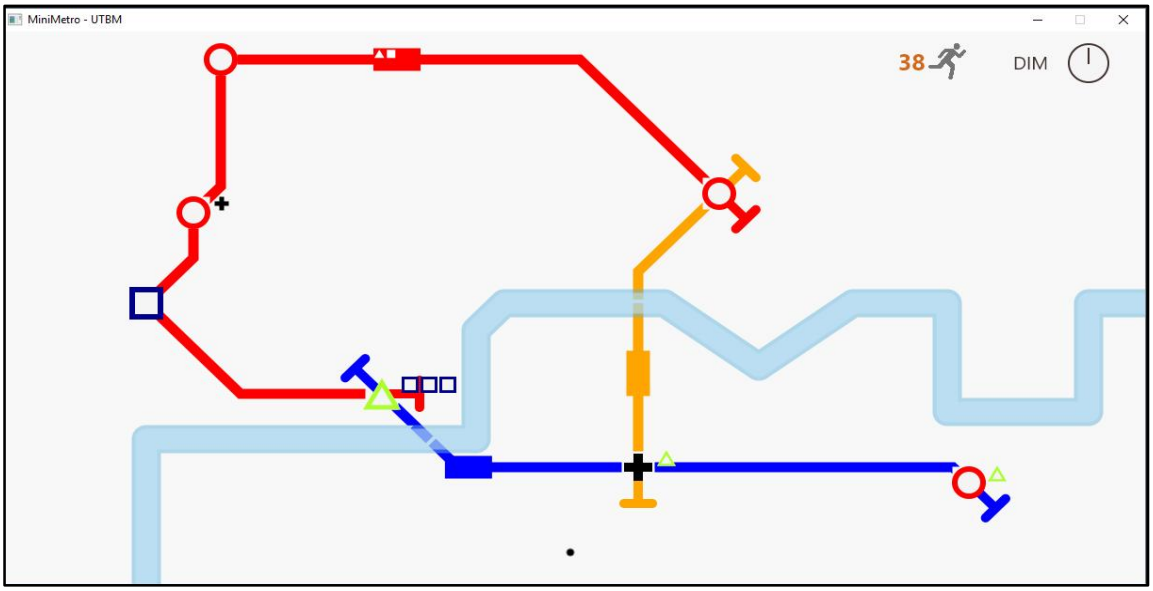
Fonctionnalités implémentées :

En essayant de nous rapprocher au plus de MiniMetro voici la liste des fonctionnalitées:

- Traçage des lignes en draggant à la souris, on peut relier plusieurs station en maintenant le drag
- Traçage des lignes en draggant une ligne.
- Suppression/traçage de lignes en appuyant sur les formes en "T" à chaque extrémitée de ligne
- Possibilité de boucler une ligne
- Gestion des tunnels, représentés dans la vue par des pointillés.
- Affichage d'une horloge, avec les jours qui défilent
- Gestion des intersections avec la rivière et les autres lignes
- Propositions de bonus toutes les semaines
- Affichage des bonus en bas de la fenêtre avec animation
- Affichage du nombres de personnes transportée en haut à droite de la fenêtre
- Les trains apparaissent graphiquement, les clients sont représentée à l'intérieur de ceux ci quand ils rentrent
- Les clients trouvent le chemin le moins long menant à leur type de stations.

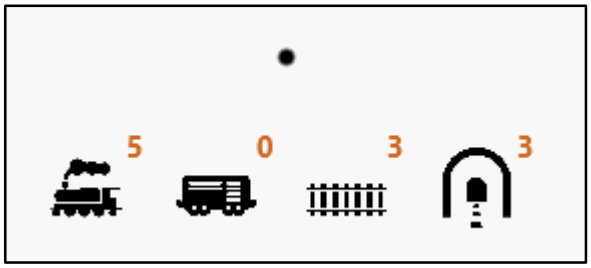
- Apparition de stations de façon aléatoire mais contrôlée : les stations n'apparaissent pas au même endroit ni dans les rivières et respectent un espacement minimal
- Apparition de clients de façon aléatoire contrôlée : un client ne va pas apparaître si sa station de destination n'existe pas encore
- Gestion du nombre de personne dans la station, menant à la défaite de la partie, avec un timer représenté par des arcs de cercles qui se remplissent et se vident
- Possibilitée de mettre le jeu en pause en cliquant sur l'horloge (avec animation), et de continuer les travaux sur notre ville
- Possibilitée de redémarrer une partie lorsque nous avons perdu
- La vue des wagons n'a pas encore été implémenté graphiquement

Aperçu de l'interface :

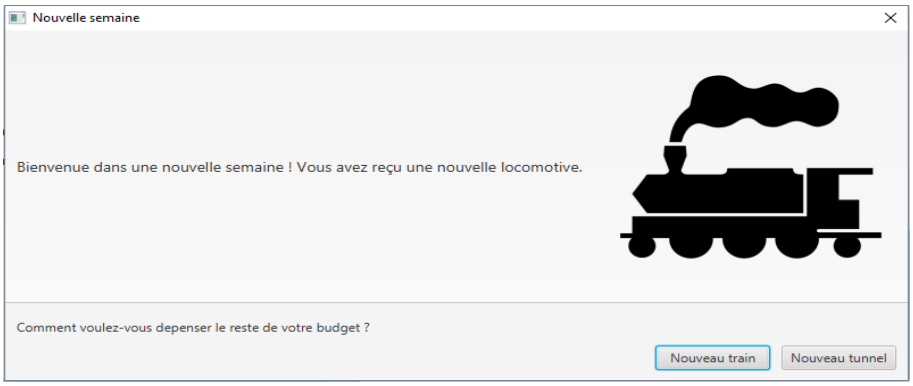


Exemple d'une partie en cours

Indicateur du nombre de passagers transportés, jours de la semaine, horloge et fonctionnalitée pause lors du clic sur cette dernière.

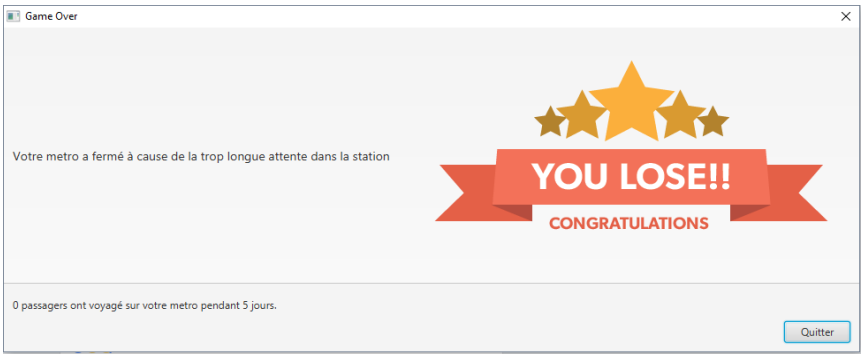


Indications des améliorations en stock lors du passage de la souris sur le point noir



Réception d'un cadeau chaque semaine et choix d'une seconde amélioration.

Chaque semaine une locomotive est offerte. Sont ensuite sélectionnés aléatoirement deux autres cadeaux parmis ligne, tunnel, wagon et train. L'utilisateur en sélectionne un parmis les deux qui s'affichent sur les boutons. C'est une fenêtre dite " MODAL ", c'est à dire qu'on ne peut pas continuer notre partie tant que nous n'avons pas choisi une amélioration. Le jeu se met en pause en attendant.



Ecran de défaite

Lorsque que le joueur perd un écran de défaite indiquant le nombre de passagers transportés et la durée de cette partie.

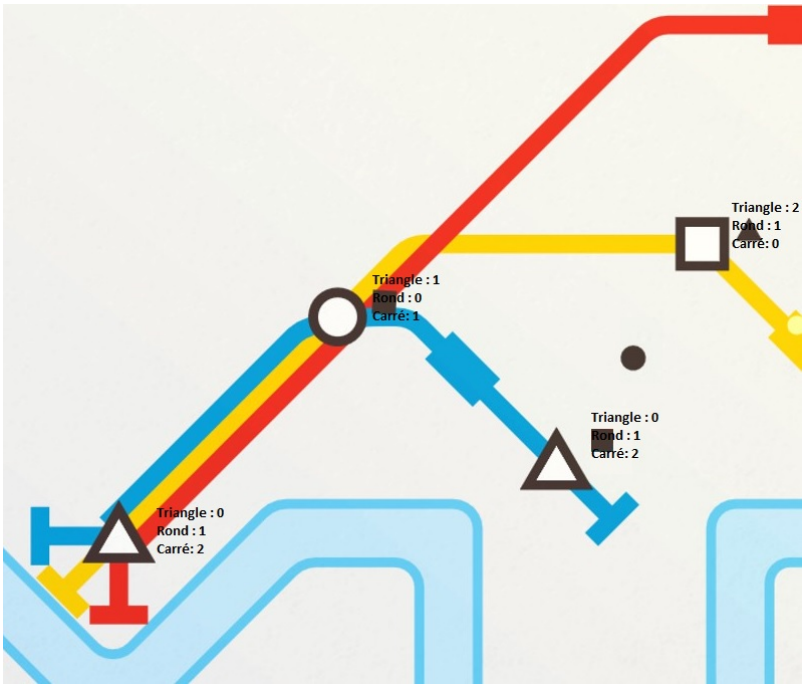
Déplacements des clients

Nous avons tout de suite repéré la difficulté du projet qui consistait à déplacer les clients. Nous donc choisi d'implémenter d'abord cette fonctionnalité et avons choisi une façon simple de gérer les déplacements des clients, lorsqu'un train arrive à une station, si la ligne du train contient une station du même type alors le client entre dans le train.

Sinon si le train s'approche (distance en termes de nombre de liens entre stations, pas de distance euclidienne) de la station la plus proche du type de station d'un client, alors ce client entre. Nous avons donc choisi de se différencier sur ce point avec le jeu original, en effet, dans ce dernier le client ne cherche jamais à savoir le plus court chemin.

Si aucun des deux cas n'est vérifié, le client n'entre pas, car il n'existe pas de chemin menant à une station de son type.

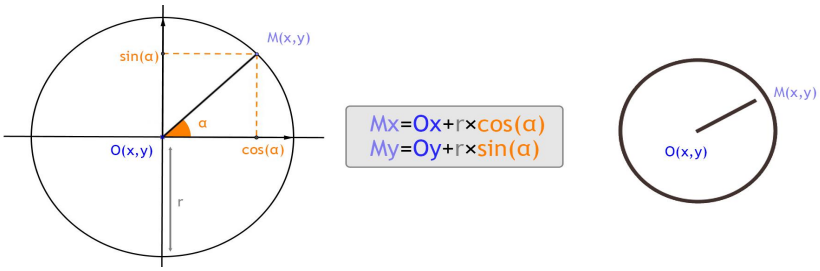
Voici un petit schéma facilitant la compréhension des distances utilisés pour notre projet :



Processus autonomes et threads :

Lors du déroulement d'une partie, plusieurs tâches sont exécutées en parallèles : il y a tout d'abord l'écoulement du temps, l'apparition de nouveaux client, l'apparition de nouvelles stations qui doivent se faire en parallèle du reste du jeu comme le mouvement des trains par exemple. Nous avons alors utilisé les Threads pour ces exécutions en parallèle en implémentant la méthode run().

Le premier Thread concerne l'écoulement du temps. Il permet d'incrémenter à intervalle réguliers grâce à la méthode sleep() le temps ainsi que le nombre de jours. C'est ici aussi que les mouvements de l'aiguille de l'horloge sont contrôlés. Pour animer, l'aiguille nous changeons la position du point situé à l'extrémité de l'aiguille par l'intermédiaire d'un petit calcul de trigonométrie.



Lors de l'apparition d'une boîte de dialogue demandant le choix d'une amélioration ou lors de la fin de la partie nous souhaitions mettre en pause le jeu comme de le MiniMetro original. Nous avons alors utilisé la méthode synchronized pour suspendre proprement le déroulement du Thread. Lors de la reprise, la méthode notifyAll() permet de débloquer le Thread.

L'apparition aléatoire de clients est réalisée grâce à un thread dont la durée de pause varie. En effet, nous passons en paramètre de la méthode sleep() de ce Thread un nombre aléatoire. Ainsi, l'apparition de nouveaux clients intervient en parallèle du reste de l'application et à intervalles aléatoires. L'apparition de nouvelles stations de manière aléatoire suit le même principe.

Lorsque le nombre de client qui attendent à une station dépasse la capacité de la station, une animation avec un cercle qui se forme autour de la station est lancé. JavaFX propose les classe Timeline, KeyValue et KeyFrame qui nous permettent d'incrémenter ou décrémenter de manière fluide la taille de l'arc de cercle autour de la station sans que nous ayons besoin de créer nous même un Thread.

Lorsque l'animation est terminée ce qui signifie que le cercle est rempli, l'utilisateur a perdu, la partie est terminée.

Conclusion

L'utilisation des diagrammes UML pour modéliser le projet nous a permis dans un premier temps de définir un cahier des charges à l'aide du diagramme des cas d'utilisations. Cette étape a été importante pour la coordination de notre travail en groupe ainsi que pour poser les bases de notre application.

En s'appuyant sur ce diagramme, nous avons pu définir les interactions entre l'acteur principal (le joueur) et notre application sous la forme de plusieurs diagrammes de séquences. Ces diagrammes représentent les cas d'utilisations les plus complexes afin de nous guider lors de l'implémentation du projet.

Nous avons ensuite créé un diagramme de classe permettant de visualiser rapidement le modèle du projet afin de pouvoir réfléchir sur le fonctionnement (controleur) de notre application, dans le but d'éviter les problèmes de modélisations pouvant survenir durant l'implémentation. Nous avons aussi détaillé des aspects plus techniques, concernant la question de la gestion des déplacements des clients, qui semble être la difficulté du sujet.

La réalisation des diagrammes UML a donc été pour nous un outil essentiel pour organiser notre projet en groupe et visualiser la conception de notre application, ceci étant incontournable pour développer un projet s'appuyant sur la programmation orientée objet.