RAILROAD INK

DOSSIER RÉTROSPECTIF

Equipe :

Gabriel Kreutser

Matheo Leroy-Chatelain

Yaroslava Khrabrova

Wissal-Asma Harrat

Enseignant:

Matthieu Dabrowski

2024-2025

I. Introduction

Pour le semestre six de notre licence nous avons eu pour projet de développer des outils visant à permettre au club Fantastiques Informatiques et Ludiques de pouvoir organiser un tournoi sur le jeu RailRoad Ink. C'est à partir de ce moment que nous avons planifié les différents modules que nous souhaitons implémenter, ainsi que tout ce qui s'y réfère.

Ce rapport vise à vous expliquer les différents modules que nous avons développés ainsi que les choix et les difficultés de conception auxquels nous avons dû faire face.

II. Travail en groupe

L'objectif de notre projet était d'avoir une organisation qui permettait à chacun de nous de participer au développement de tous les modules en travaillant à plusieurs. Pour cela, il à fallu trouver un système de communication centrale sur lequel on pourrait se reposer pour échanger et organiser le développement de notre projet. C'est pour cela qu'on à créé, dès la première semaine, un serveur Discord qui lui était dédié.

En ce qui concerne notre répartition des tâches, nous avions chacun des rôles durant ce projet. Mathéo était notre chef de la section client, Yaroslava était notre chef de la section algorithme et modélisation, Gabriel était notre chef de la section debug.

Nous avons déduit ces rôles à partir d'une tendance que nous avons observée dans la répartition des tâches. Au début du projet, nous n'avons pas souhaité définir ces rôles car nous voulions plutôt laisser les membres du groupe travailler sur les fonctionnalités qu'ils souhaitaient. Mais au fur et à mesure, nous avons remarqué cette fameuse tendance dans le groupe en fonction de la répartition des fonctionnalités.

Enfin, pour s'assurer que notre développement soit efficace et simultané, nous avons décidé de diviser les différents modules en plusieurs dépôts Git. Cela nous permettait de ne pas mélanger les fonctionnalités que nous développions. Nous avons aussi utilisé des branches pour nous permettre de travailler à plusieurs sur le même module sans se soucier des problèmes de conflit. Une fois que le code était bien avancé sur plusieurs branches, on se réunissait et on effectuait une merge request afin de rassembler le code du module sur la branche principale du dépôt, et ainsi repartir sur une nouvelle fonctionnalité pour ce module.

Mais cette organisation peut malheureusement présenter des failles à certains moments. Par exemple, il nous est arrivé de nous retrouver face à un problème de version sur un de nos dépôts suite à une mauvaise merge request. Pour faire face à ce problème, Gabriel à réussi à rassembler les dernières versions des différents fichiers à l'aide des précédents commits.

On peut donc dire que malgré cet incident, nous sommes persuadés que notre organisation était la bonne au vu des résultats qu'elle à apporté vis-à-vis de l'avancement du projet.

III. Généralité

Pour ce projet, on a décidé d'opter pour une architecture par microservices (MOM). Si on regarde la manière dont le projet nous est présenté, cette architecture nous paraît la plus logique. Le fait que nous devions utiliser un réflecteur pour faire transiter tous nos messages implique de devoir utiliser des modules distincts les uns des autres.

Un autre avantage de cette architecture est la palette de langages pouvant être utilisés de manière simple car les modules ne vont communiquer entre eux que via le réflecteur. Ils vont utiliser un protocole de communication interne identique. On à donc essayé de varier les langages utilisés dans les différents modules, tels que le Java pour l'administrateur, Haskell pour le client manuel ou encore Python pour l'intelligence artificielle. On a aussi choisi des frameworks précis afin de faciliter le développement du projet. On peut citer JavaFX par exemple pour tout le côté Front-End, et même Express pour pouvoir gérer notre interface Web.

Pour développer ce projet, nous avons décidé de mettre en place une intégration continue, afin de voir, grâce à nos tests, que notre projet fonctionne peu importe les changements ajoutés. Nous pouvons maintenant parler plus précisément des modules et de leur fonctionnalité.

IV. Administrateur

L'administrateur est responsable de la gestion globale du jeu et du respect strict des règles. Son rôle principal est de superviser l'intégralité du processus de jeu, depuis l'admission des joueurs jusqu'à la fin de chaque round.

L'administrateur agit comme un client qui interprète les messages reçus grâce à une classe Interpréteur. L'Interpréteur va ainsi étudier le message reçu et l'envoyer vers les interprètes de commandes comme *ENTERS*, *ELECTS*, et autres. Ces interprètes de commandes représentent parfaitement toutes les commandes demandées, mais nous avons également ajouté des commandes comme *PLAYS* ou *SCOREROUND* grâce à l'ajout de notre propre réflecteur que nous verrons plus tard dans la présentation des modules.
Pour permettre à notre projet d'être au maximum extensible, nous avons utilisé le système de réflexion dans notre interpréteur. Cela permet d'exécuter la classe liée à cette commande de manière autonome. Ce système va exécuter la méthode statique *execute* de la classe correspondant à l'action demandée. Ainsi, si on décide d'ajouter de nouvelles actions, il n'est pas nécessaire de modifier l'interpréteur car celui-ci trouvera automatiquement la classe associée à cette nouvelle action.

Pour la création de cet administrateur, nous avons procédé par une division des tâches. Mathéo a commencé le développement de notre administrateur par la création du client, de l'interpréteur, du parser ainsi que de la classe Messages, permettant ainsi d'interpréter plus facilement les messages grâce à cet interpréteur.

Gabriel et Yaroslava ont poursuivi notre Administrateur. Nous avons ainsi recodé le jeu en entier afin d'avoir une structure solide pour poser les bases de l'administrateur.

Dans cette structure, l'administrateur utilise le jeu pour étudier les requêtes des autres joueurs. Ainsi, lorsqu'un joueur utilise la commande *ENTERS*, il devient un joueur du jeu mais ne peut pas encore jouer. Pour qu'il puisse jouer, il doit utiliser la commande *ELECTS*, ce qui lui permet de devenir un "joueur élu". Une fois la commande *PLAYS* lancée, le jeu commence avec uniquement les joueurs élus, et les joueurs non élus ne sont pas pris en compte.

Chaque player, donc chaque utilisateur du réflecteur, possède un board qui contient lui-même des cellules où les tuiles seront placées. Suite à de grandes réflexions entre Yaroslava et Gabriel, nous avons décidé d'intégrer des nodes dans les cellules et les tuiles (que nous appelons "face").

Une Node représente un ensemble de connexions avec d'autres nœuds, formalisé sous la forme de paires (Node, Connection), ainsi que le type de connexion (road or rail). Cette structure permet de modéliser efficacement les relations entre les différents éléments du jeu. Dans notre implémentation, chaque cellule ou tuile possède cinq nodes: top, bottom, right, left et centre.

Nous avons pris la décision d'ajouter spécifiquement les nodes centrales pour permettre une gestion plus précise des gares et stations, et pour obtenir une représentation plus fine et extensible.

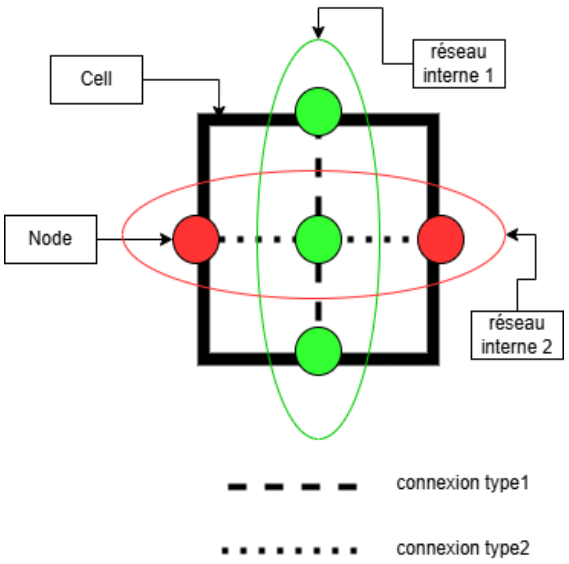

Schéma 1

Au-delà des nodes, chaque cellule possède également un *InnerConnectionsGraph*, dont l'utilité est particulièrement visible dans le cas des tuiles spéciales (voir Schéma 1). Par exemple, une tuile HR possède deux connecteurs internes. Ce graphe représente

l'ensemble des connexions internes entre les nodes pour toutes les tuiles et offre la possibilité d'intégrer facilement des extensions au système.

La relation entre cellules et tuiles est gérée de façon systématique. Les cellules et les tuiles contiennent toutes deux des nodes. Lorsqu'une tuile est placée dans une cellule, nous reprenons les mêmes patterns de connexions entre les nodes à l'intérieur de la tuile et les dupliquons dans la cellule.

Pour assurer une gestion fluide des connexions externes, les cellules voisines partagent les mêmes nodes sur leurs côtés adjacents. Plus précisément, la node gauche d'une cellule est identique à la node droite de son voisin de gauche. Cette conception permet de maintenir le réseau global.

Ce système de nodes présente également un avantage pour la gestion des rotations. En effet, il suffit de déplacer les nodes pour que toutes leurs connexions soient automatiquement ajustées, ce qui simplifie l'implémentation des mécanismes de rotation. Pour compléter cette approche, nous avons implémenté un pattern Stratégie pour chaque type de rotation. Cette architecture permet d'ajouter facilement de nouveaux types de rotations sans modifier le code existant. Cela est renforcé par le pattern factory pour la gestion des rotations suite à la réception de la commande *PLACES*.

Pour gérer les sorties du plateau, nous avons introduit des "*BorderCell*" qui possèdent une connexion dans la direction de la sortie mais également son type.

Suite à l'ajout de ces nodes, nous avons codé les règles de placement, qui suivent exactement les règles du jeu. Ces règles héritent d'une interface permettant de les modifier si besoin. Cependant, grâce à l'utilisation des nodes et aux règles actuelles, nous n'avons pas besoin de les changer lorsque nous ajoutons de nouvelles tuiles ou d'autres éléments.

Lorsqu'un joueur émet la commande *PLAYS*, cela déclenche l'exécution d'un thread dédié qui invoque la méthode play de la classe *Game*. Cette conception permet de maintenir une interaction continue, autorisant la réception et le traitement de messages en simultané avec l'avancement du jeu. La classe game constitue l'architecture centrale du système, hébergeant la liste des joueurs précédemment admis via la commande *ENTERS*, ainsi que la gestion des rounds et du système de score.

La méthode *play* génère, à chaque round, des lancers de dés qui produisent des tuiles à placer. Ces tuiles sont transmises au réflecteur par la commande *THROWS*, avec une attente de la réception de son propre message. Afin de permettre une interprétation qui va conduire à la création d'un nouveau round qui va être ajouté à la classe *Game* ainsi qu'à la distribution des tuiles à l'ensemble des joueurs élus grâce à l'utilisation des patterns factory pour la création des dés et des tuiles.

Les joueurs interviennent ensuite en utilisant la commande *PLACES* pour positionner leurs cellules. Le système procède à une interprétation de chaque placement, validant l'emplacement proposé ou non. Un placement considéré comme valide est immédiatement accepté, tandis qu'un placement inapproprié déclenche l'envoi d'une commande *BLAMES*, incluant le rang du message incriminé. Suite à cette phase de placement, l'administrateur

émet la commande *SCORES*, communiquant les différents types de scores générés par le coup précédent.

Le round se termine lorsque tous les joueurs ont placé leurs 4 tuiles qui leur ont été distribuées, ou bien qu'ils ont décidé de passer leur tour avec la commande *YIELDS* les mettant ainsi dans l'état temporaire *isPassRound*.

Pour calculer les scores, nous avons développé une bibliothèque dans un module à part de graphes complète qui inclut plusieurs classes essentielles : Graph, Edge, Connectivity Inspector (analyse des composantes connexes présentes dans le graphe), Dijkstra Algorithm (les calculs de distance et l'identification des chemins optimaux).

Notre implémentation se distingue par une approche de modélisation des graphes. Nous n'utilisons pas simplement les nodes ou les cellules comme sommets du graphe, mais représentent plutôt les connexions internes/externes comme sommets. Cette méthode facilite la division des réseaux et permet une prise en compte des tuiles spéciales, comme les tuiles HR. Elle offre également une représentation plus précise des relations entre les éléments du jeu. Dans cette logique, le graphe du plateau représente les ensembles de nodes connectées, où les connexions internes deviennent des connexions externes dans le graphe global.

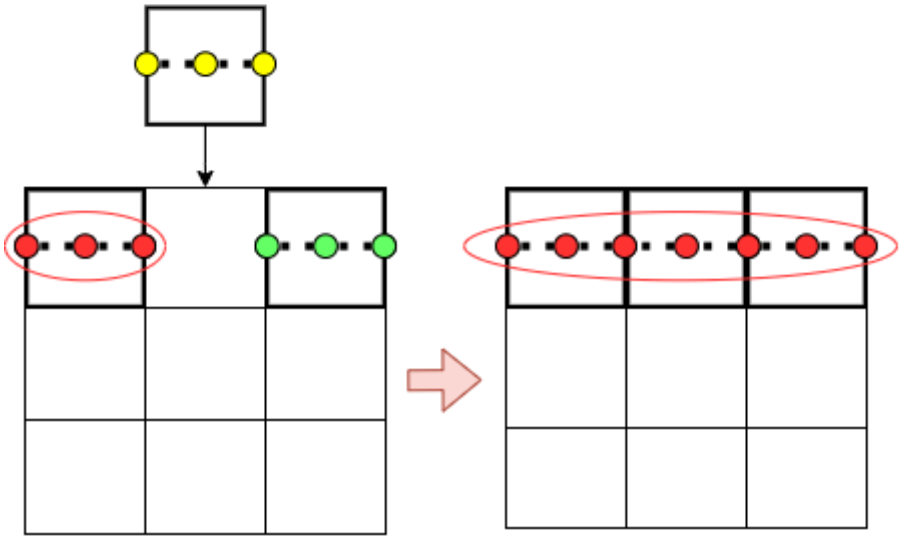

Schéma 2

Pour gérer efficacement la complexité du jeu, l'ajout d'une face dans une cellule entre deux cellules où des faces sont déjà présentes constitue un défi technique important (voir Schéma 2).

Pour résoudre ce problème, nous avons implémenté le pattern Observer. Ce design pattern notifie automatiquement le plateau lors de l'ajout d'une nouvelle tuile et déclenche le recalcul des connectivités externes dans le graphe représentant le plateau.

En ce qui concerne les algorithmes de calcul, pour déterminer le chemin le plus long, nous utilisons un parcours en profondeur *(DFS - Depth-First Search)* du graphe. Cet algorithme explore toutes les branches possibles pour identifier le chemin de longueur maximale. De façon complémentaire, la détection des réseaux s'appuie sur un algorithme d'identification des composantes connexes dans un graphe non-orienté, permettant d'identifier efficacement les ensembles de nodes formant des réseaux distincts.

Notre système de calcul de score utilise une architecture basée sur le pattern Strategy. Au cœur de cette architecture se trouve l'interface *ScoreCalculator* qui définit le contrat commun pour tous les calculateurs de score spécifiques (méthode de calcul +bi- fonction). Chaque type de calcul de score est implémenté dans une classe dédiée qui hérite de cette interface.

Au cœur de notre système de calculs se trouve le *Score Manager*. Il calcule le score total et maintient une map des différents calculateurs de score. Son architecture supporte à la fois le mode classique et les extensions du jeu. La conception modulaire permet d'ajouter facilement n'importe quel type de calculateur de score, comme des bonus. Chaque calculateur possède une bi-fonction qui définit comment son résultat s'intègre dans le calcul du score total. Par exemple, le résultat du calcul des fausses routes est soustrait du total.

Notre implémentation va au-delà des fonctionnalités de base et gère plusieurs extensions et cas particuliers. Le calcul des plus longs chemins inclut toutes les sous-branches pour une évaluation complète. Les stations sont considérées comme des arêtes pour le calcul du chemin total. Le système prend en compte les fausses routes, y compris celles touchant le bord du plateau.

Tout au long du développement, nous avons rencontré plusieurs défis techniques majeurs. La gestion de l'ajout d'une tuile à n'importe quel endroit du plateau a été résolue grâce à l'implémentation du pattern *Observer*. La détermination du réseau auquel une node doit être rattachée utilise le *Connectivity Inspector*.

Enfin, la gestion complexe du score pour les chemins les plus longs, notamment avec les nodes centrales, s'appuie sur des algorithmes spécialisés qui prennent en compte les configurations particulières.

Nous avons également fait un bon nombre de tests afin de s’assurer de la maintenabilité du code. Nous avons ainsi ***87%*** de ligne couverte et ***71%*** de mutation testées. Ce sont Gabriel, Wissal et Yaroslava qui se sont occupés de réaliser les tests.

V. Interface graphique

Pour nous permettre de jouer et de visualiser les différentes parties en cours, nous avons décidé de développer une interface graphique via un site web développé sur le framework *NodeJS* avec *Express*. C’est Mathéo qui s’est occupé de développer cette partie.

Lorsque vous arrivez sur la page principale, vous avez deux possibilités : jouer ou visualiser la partie en cours. Si vous décidez de jouer, vous devrez entrer un identifiant avant de pouvoir lancer la partie quand vous le souhaitez. Mais si vous décidez de visualiser une

partie, vous aurez alors accès à tous les plateaux des joueurs présents et vous pourrez voir leurs coups ainsi que leurs scores en temps réel.

Pour développer cette interface, on s'est reposé sur le principe de l'administrateur. C'est-à-dire un client qui se connecte au réflecteur et qui reçoit la totalité des messages, et qui les traitent un par un via un interpréteur.

Pour cette interface, on a décidé de faire certains choix de conception. Pour la partie de jeux, on à décidé de ne pas implémenter les règles de placement, car on ne voulait pas avoir à les développer dans un nouveau langage sachant qu'elles existaient déjà dans l'administrateur. On s'est donc basé sur le système de blâmes renvoyés par l'administrateur si jamais une tuile est mal placée.

Il en va de même pour le calcul du score. C'est l'administrateur qui envoie le score de chaque joueur à chaque coup, et non l'interface qui le calcul elle-même. L'objectif était d'alléger le code de l'interface et d'éviter de multiplier les codes réalisant les mêmes fonctionnalités entre les différents modules.

VI. Système de persistance

Le module de persistance, implémenté en Python, offre une solution pour l'archivage des messages échangés via WebSocket. Développé par Wissal, ce composant assure une journalisation fiable des conversations tout en filtrant les données non pertinentes. Sa conception minimale mais efficace en fait un élément clé pour le suivi des activités du système.

Le système écoute en permanence le flux des messages transitant par le réflecteur *WebSocket*, appliquant un filtrage qui élimine les messages vides ou les commentaires *(lignes commençant par #).* Chaque message valide est ensuite enregistré dans le fichier "*messages.txt*" avec une structure claire : un message par ligne, tout en conservant son ordre d'arrivée. La gestion des connexions inclut une détection propre des déconnexions et des reconnexions automatiques.

L'architecture repose sur La bibliothèque websockets qui gère les communications réseau asynchrones, tandis que asyncio permet un traitement non-bloquant des messages. Le système utilise un mécanisme d'écriture simple mais efficace : chaque message est immédiatement écrit sur disque (en append) après une validation basique, garantissant ainsi la persistance des données même en cas d'arrêt brutal.

VII. Client manuel

Le client Haskell offre une interface fiable pour communiquer avec le réflecteur via WebSocket. Développé entièrement par Yaroslava, ce module assure une gestion des interactions temps réel avec le réflecteur tout en maintenant une cohérence avec les règles métier du système.

Le client intègre plusieurs fonctionnalités clés . La validation des messages analyse en profondeur chaque requête avant transmission, vérifiant à la fois sa structure syntaxique et sa conformité sémantique avec les règles du jeu. La gestion des états permet une transition entre les différents rôles (*Waiting*, *Active*, *Judge*, *Organizer*) bloquant les actions non autorisées. En parallèle, le système maintient une communication bidirectionnelle asynchrone, capable de traiter simultanément les envois et réceptions sans perte de performance.
Le module intègre une suite de tests complète couvrant les principales fonctionnalités

Sur le plan technique, la couche réseau utilise la bibliothèque websockets pour établir des connexions persistantes et faible latence. L'automate à états finis modélise les transitions autorisées entre les différents statuts des utilisateurs. La validation des messages s'appuie sur un système combinant parser et vérificateur de types, tandis que Stack assure la gestion cohérente des dépendances et la reproductibilité des builds.

Lors du développement, une des principales difficultés a été l'utilisation du gestionnaire de projet Stack, qui nécessitait une prise en main spécifique pour la gestion des dépendances et la compilation du projet. De plus, l'intégration de bibliothèques peu familières, comme WebSocket, a représenté un défi supplémentaire, demandant une phase d'apprentissage et d'expérimentation pour assurer une communication stable et efficace entre le client et le réflecteur.

VIII. Arbitre manuel

Dans le cadre de notre projet, Gabriel a développé notre arbitre manuel en Python qui permet une interaction asynchrone via un WebSocket à notre réflecteur.

Les fonctionnalités de cet arbitre sont centrées sur la communication entre l'arbitre et le réflecteur. Cet arbitre permet d'envoyer et de recevoir des messages en temps réel, avec une identification unique. L'utilisateur peut saisir des messages qui sont immédiatement transmis, tout en recevant simultanément les réponses du serveur. Tout cela permet à une vraie personne de se connecter avec cet arbitre manuel et de vérifier manuellement si oui ou non, les coups sont acceptables, ou bien donner le score ainsi que toutes les autres possibilités qu'un arbitre peut faire.

L'implémentation technique repose sur l'utilisation des bibliothèques asyncio et websockets de Python, qui permettent de gérer efficacement les opérations asynchrones. L'utilisation des méthodes asynchrones permettent de traiter simultanément l'envoi et la réception de messages, permettant ainsi d'envoyer des messages au réflecteur tout en continuant d'en recevoir.

IX. Interface graphique de la persistance

Cette partie, développée par Gabriel et Yaroslava, vise à créer une interface graphique ayant pour but d'analyser et de visualiser des données de jeu, permettant aux utilisateurs d'explorer et d'interpréter la persistance des sessions de jeu.

En termes de fonctionnalités, notre interface est constituée de trois modules principaux. Le module de parsing se charge d'analyser les fichiers de persistance pour alimenter notre structure de données. Nous offrons également des options de sélection permettant aux utilisateurs de cibler précisément les informations qu'ils souhaitent visualiser, comme les rounds, les parties ou bien les joueurs, avec un affichage qui s'adapte à ces choix.

L'affichage du plateau propose une visualisation dynamique et différenciée des chemins, réseaux et routes correspondant à chaque type de calcul de score, qui permet aux joueurs d'identifier clairement les éléments contribuant aux différentes composantes de leur score. Le module setting nous propose quant à lui des options de personnalisation, incluant le style d'interface et l'activation ou non de la musique d'ambiance. Tous ces choix s'appliquent à l'ensemble des scènes de l'application. Enfin, le module analytique permettant de charger les données du fichier de la persistance, d'examiner en détail les informations relatives à un joueur ou une partie spécifique via leur identifiant, et de générer une visualisation des données analytiques sous forme de graphiques.

Concernant l'implémentation technique, pour le chargement des données, nous avons développé un parser basé sur un automate à états finis en utilisant le design pattern state.Le parser est capable d'analyser les fichiers de persistance. Les données extraites sont ensuite organisées dans une classe dédiée (GameData), facilitant leur utilisation pour l'affichage dans le javafx. Nous avons implémenté plusieurs classes-détecteurs spécialisées qui permettent d'identifier et de récupérer les cellules, chemins ou routes spécifiquement concernées par chaque type de calcul de score particulier.

Notre système exploite diverses bibliothèques de manipulation graphique capables de modifier dynamiquement les images au format SVG en temps réel. Cette capacité nous permet d'ajouter des éléments visuels distincts comme des surlignages colorés pour les chemins les plus longs, des teintes différenciées pour les réseaux connectés ou des marqueurs spécifiques pour les fausses routes identifiées par le système de scoring. Pour notre implémentation des interfaces graphiques en JavaFX, nous avons adopté une approche modulaire en attribuant une scène distincte à chaque page de l'application.

Chaque scène est elle-même décomposée en composants répartis dans différentes classes, ce qui améliore considérablement la maintenabilité du code et la lisibilité de l'architecture. Le module setting repose sur la classe *SettingsData* qui récupère l'ensemble des réglages applicables aux différentes scènes, permettant ainsi de modifier directement les caractéristiques ciblées sur toutes les scènes. Pour la partie analytique, nous avons conçu une classe *AnalyticsData* qui structure les informations après leur parsing, rendant possible leur utilisation optimale pour générer les différentes représentations graphiques nécessaires à l'analyse.

Au cours du développement, nous avons rencontré plusieurs difficultés techniques. Le traitement des images au format SVG a constitué un défi majeur,car nous avions besoin d'utiliser les SVG afin de visualiser le plateau des joueurs, afin de ne pas mettre une grille au-dessus d'une image en png. Mais cela a entraîné des complications pour la rotation des tuiles à placer. Cette problématique a été résolue grâce à l'intégration de la bibliothèque Batik, qui nous a permis de parser efficacement les fichiers SVG et de les convertir en objets exploitables dans notre interface.

Une autre difficulté significative concernait l'adaptation du parser pour qu'il soit parfaitement compatible avec le format de nos données de persistances. Nous avons surmonté cet obstacle en implémentant le pattern State combiné à un automate, approche qui s'est révélée particulièrement adaptée pour analyser avec précision la structure de nos fichiers de persistances.

X. IA de l'interface graphique

Notre interface graphique intègre une "intelligence artificielle" dont la principale fonctionnalité est de reprendre les mêmes tuiles lancées pendant un round afin de visualiser les solutions optimales qu'elle aurait pu réaliser. Cette fonctionnalité permet aux joueurs d'analyser leurs parties et de comprendre les stratégies les plus efficaces.

Pour l'implémentation de cette intelligence artificielle, nous avons développé deux modèles distincts. Le premier modèle, celui que nous utilisons actuellement dans l'application, repose sur un algorithme que nous avons conçu. Son fonctionnement s'articule autour de la récupération d'informations sur toutes les possibilités de placement pour chaque tuile, en utilisant la classe *PossibilitiesForAllFaces*. Il procède ensuite à une imitation récursive des différentes possibilités tout en évaluant le potentiel futur score de chaque coup. Pour déterminer la valeur d'un placement, l'algorithme s'appuie sur les calculateurs de scores comme référence, ce qui lui permet d'identifier les solutions les plus prometteuses.

Le second modèle, développé parallèlement, implémente l'algorithme classique du minimax. Ce dernier intègre la notion de profondeur qui définit le nombre de niveaux de recherche dans l'arbre des possibilités. Il explore différentes combinaisons pour l'ordre de placement des tuiles. Il crée et analyse des branches d'exploration, et retourne finalement le mouvement offrant le score optimal pour chaque branche évaluée.

Lors du développement de ces intelligences artificielles, nous avons rencontré plusieurs difficultés significatives. La première concernait l'algorithme lui-même et particulièrement le temps de chargement considérable nécessaire pour explorer l'ensemble des possibilités. Pour résoudre ce problème, nous avons mis en place une stratégie de réduction du champ des possibilités en introduisant un coefficient de 80% du meilleur score obtenu. Cette approche nous permet de ne considérer que les coups véritablement prometteurs et d'explorer les étapes suivantes que si le coup initial présente un potentiel suffisant, réduisant ainsi drastiquement le temps de calcul.

Une autre piste explorée, mais non réalisée en raison de contraintes pratiques, concerne l'implémentation d'une IA basée sur le deep learning. Cette approche s'est heurtée au

problème du manque de données structurées nécessaires pour mettre en place un apprentissage efficace, nous amenant à privilégier notre solution algorithmique.

XI. IA

Pour parfaire notre projet, nous avons décidé de développer une IA pour pouvoir jouer à RailRoad Ink. C'est Mathéo qui s'est occupé de cette partie.

Pour la développer, on s'est reposé sur un algorithme MCTS (Monte Carlo Tree Search). L'objectif de cet algorithme est de trouver les meilleurs coups possibles à chaque round en simulant plusieurs parties.

L'algorithme MCTS se divise en 4 parties différentes :

1. Sélection

Le but de cette partie est d'explorer l'arbre de manière intelligente en choisissant le meilleur nœud possible. Pour cela, on descend dans l'arbre depuis la racine jusqu'à atteindre un nœud non exploré. A chaque nœud visité, il choisit le meilleur enfant possible en utilisant la formule *UCB1* (*Upper Confidence Bounds*) jusqu'à tomber sur un nœud non visité.

Si on regarde plus en détail à quoi correspond la formule *UCB1*, on à :

$$UCB1 = \overline{X_i} + c\sqrt{\frac{\ln N}{n_i}}$$

- *Xi* est la moyenne des récompenses obtenues à l'action i
- *N* est le nombre d'essais réalisés
- *ni* est le nombre de fois ou la branche à été sélectionnée
- *c* est un paramètre pour régler l'équilibre entre exploration et exploitation

Exploitation : La première partie (Xi) favorise les actions qui ont données des bonnes récompenses (en l'occurrence les coups avec le meilleur score).

Exploration : La deuxième partie favorise l'utilisation de nœuds non visités.
A ce moment-là, on passe à la deuxième phase.

2. Expansion

Cette partie vise à ajouter de nouveaux nœuds à l'arbre. Quand on tombe sur un nœud non visité, on ajoute tous les nœuds possibles sous forme de nouveaux nœuds.

3. Simulation

On simule jusqu'à la fin de la partie les coups possibles, et on obtient un score final.

4. Rétropropagation

On remonte jusqu'à la racine en mettant à jour le nombre de simulations effectuées et le score moyen obtenu pour permettre d'avoir de meilleures décisions aux tours suivants.

Si on applique cet algorithme à notre IA on obtient la structure suivante :

- Sélection : C'est la fonction ***best_child*** qui s'occupe de réaliser cette partie du code. Elle sélectionne le nœud enfant le plus prometteur en utilisant la formule UCB1 et descend jusqu'à arriver à un nœud non visité.
- Expansion : C'est la fonction ***expand*** qui réalise l'expansion de l'arbre à partir du nœud sélectionné précédemment.
- Simulation : Cette partie est écrite dans la fonction ***mcts_search*** qui va sélectionner un coup à réaliser et l'évaluer.
- Rétropropagation : C'est la fonction ***update*** qui va réaliser cette partie de l'algorithme. Elle met à jour la valeur et le nombre de visites du nœud.

Malheureusement, l'IA n'est pas aussi performante que l'on ne l'aurait souhaité. L'algorithme MCTS demande un grand nombre d'itérations pour fonctionner. Dans notre algorithme nous avons établi un nombre de simulation équivalent à 1000. Cela lui permet de chercher les meilleurs coups sans prendre trop de temps.

De plus, ne connaissant pas les tuiles qui seront tirées au round suivant, l'IA essaye de prévoir en connectant un maximum de sortie mais cela est difficile car elle n'a pas toujours les bonnes tuiles à placer. Il arrive donc qu'à certains moments ses décisions ne soient pas les meilleures possibles. Pour pallier cela, on pourrait développer une IA se basant sur les réseaux de neurones, ou bien améliorer l'algorithme MCTS pour le rendre plus performant en essayant de garder un temps d'exécution convenable.

XII. Reflecteur

Nous souhaitions aussi pouvoir étendre notre projet en envoyant de nouvelles commandes au réflecteur. C'est pour cela que nous avons produit le nôtre en utilisant le langage Rust. Nous avons choisi ce langage pour sa rapidité, ce qui nous assure que face à un nombre important de messages, le réflecteur reste performant. C'est Mathéo qui s'est occupé de développer ce module du projet.

Notre réflecteur fonctionne exactement comme celui qui nous a été fourni au début du semestre. Il accepte des connexions, vérifie les messages (notamment *ENTERS* et *ELECTS*). Il les stocke dans une liste et les renvoie à toute personne qui se connecte. Il envoie aussi le message *LEAVES* lors d'une déconnexion. Un certain nombre d'actions de base sont incluses dans le réflecteur, mais on peut en ajouter de nouvelles via l'option *--add* précisée au lancement du réflecteur.

Pour le développer on s'est reposé sur la bibliothèque ***tokio*** qui permet de gérer les connexion websocket. Son fonctionnement se divise en plusieurs étapes.

Tout d’abord, le serveur commence par initialiser les paramètres via la bibliothèque ***clap***, qui permet de récupérer les arguments passés dans la ligne de commande. Deux arguments sont importants au lancement du réflecteur : le port d’écoute et les nouvelles actions.
Par défaut, il écoute sur le port 8000, sauf si un autre est spécifié via l'argument *--port*.

Une fois la configuration terminée, le serveur crée plusieurs structures de données pour gérer les clients ainsi que les messages. On utilise un HashSet pour stocker les identifiants des clients connectés. On utilise une autre liste pour conserver un historique des messages envoyés afin de pouvoir les envoyer aux nouveaux clients qui se connectent au réflecteur.
Et enfin on utilise un canal de type broadcast pour pouvoir transmettre les messages reçus à tous les clients.

Lorsqu’un message est reçu d’un client, il est d’abord analysé et validé. On vérifie que l’auteur du message a bien envoyé le message *ENTERS* avant. Si les conditions sont remplies, le message est stocké et retransmis à tous les autres clients. Mais si les conditions ne sont pas remplies, le client est immédiatement déconnecté.
Enfin, lorsqu’un client se déconnecte (manuellement ou automatique), son identifiant est retiré de la liste des clients et un message *LEAVES* est envoyé à tous les clients.

XIII. Conclusion

Ce projet nous a permis de concevoir une plateforme complète pour organiser des tournois de « RailRoad Ink », intégrant un administrateur central, des interfaces utilisateurs, un système de persistance et une IA. Malgré les défis techniques liés à la modélisation des règles et à la coordination entre modules, nous avons su les surmonter grâce à une architecture microservices flexible et l'application rigoureuse de design patterns vus au semestre dernier. Si une approche client-serveur aurait pu simplifier certaines interactions, le choix d'une architecture MOM a prouvé sa pertinence pour ce projet multilingue.
Les compétences acquises en COO, graphes, algorithmique et gestion de projet trouvent ici une application concrète. Ce système ouvre des perspectives intéressantes, que ce soit pour ajouter des extensions de jeu, améliorer l'IA ou développer de nouvelles interfaces.

XIV. Annexe

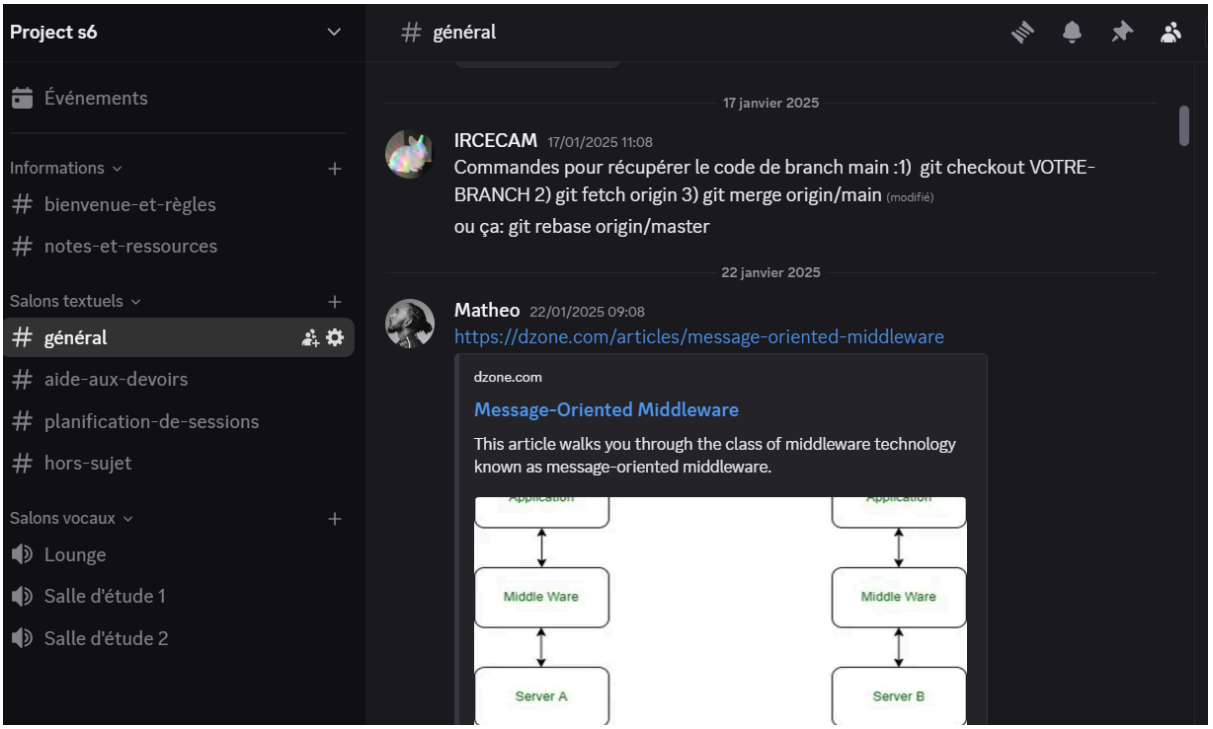

Image 1: Capture de l'écran de server Discord créé pour la communication entre les membres du groupe

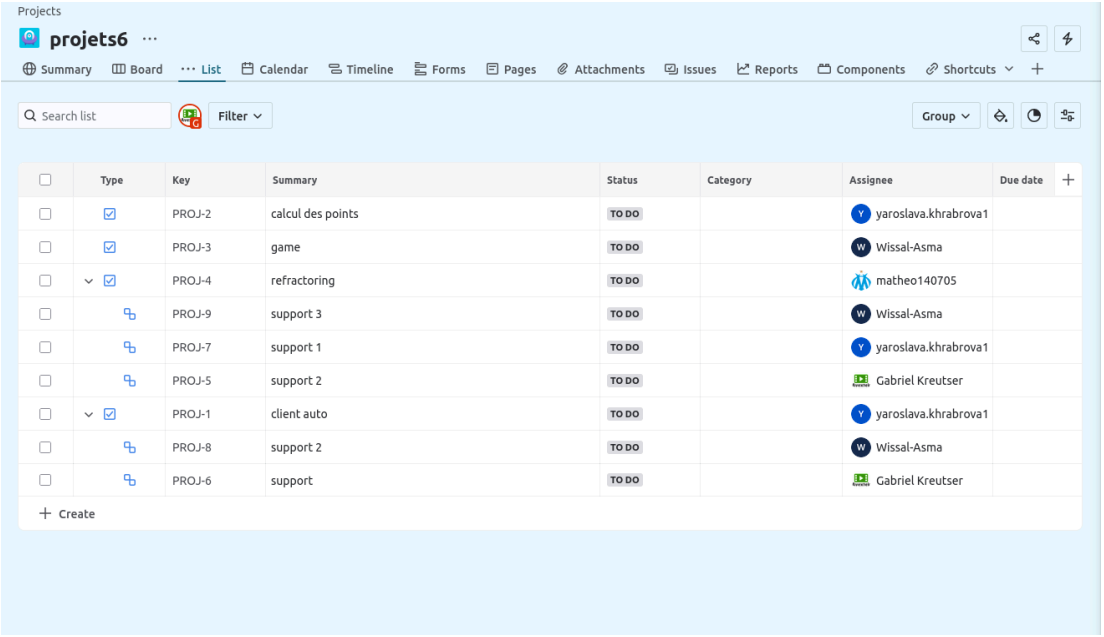

Image 2: Capture de l'écran d'outil Trello utilisé pour la répartition des tâches

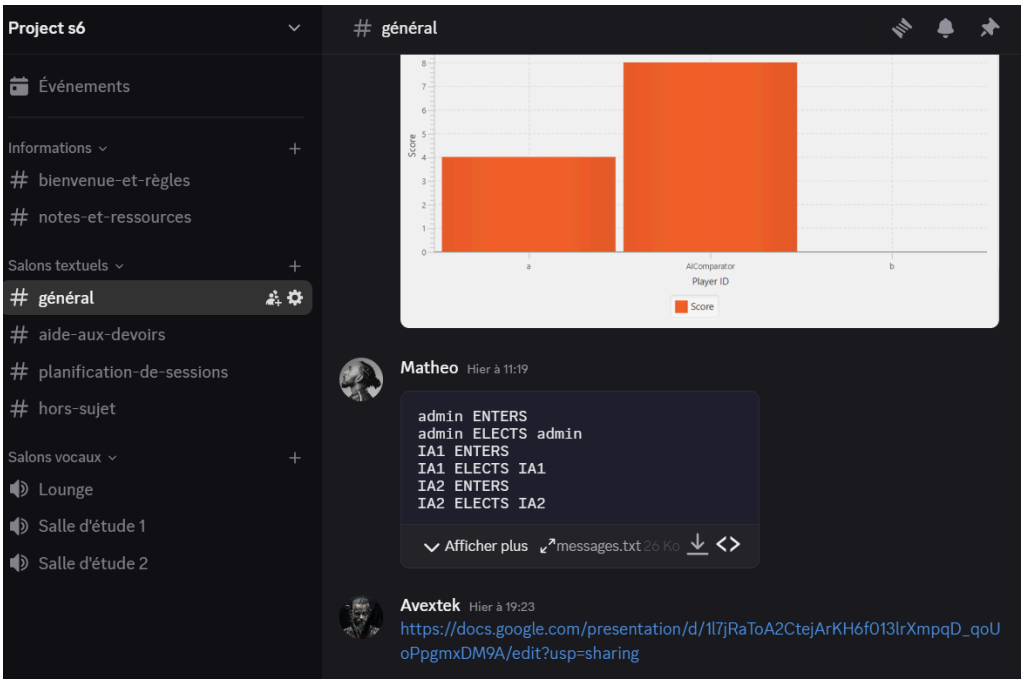

Image 3: Capture de l'écran de server Discord créé pour la communication entre les membres du groupe

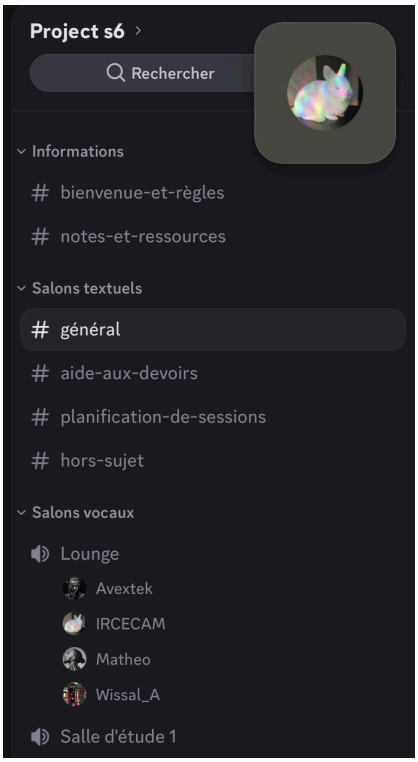

Image 4: Capture de l'écran de la communication en vocal entre les membres

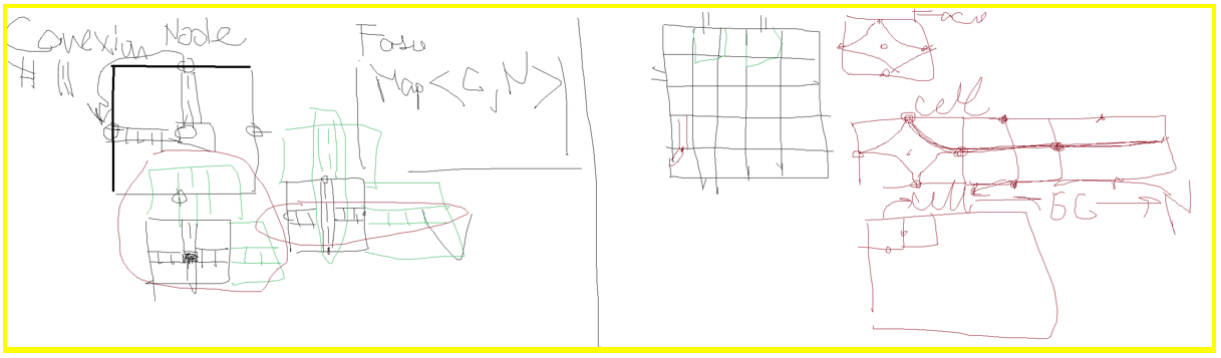

Image 5: Capture de l'écran de l'utilisation d'outil Paint pour modéliser un problème

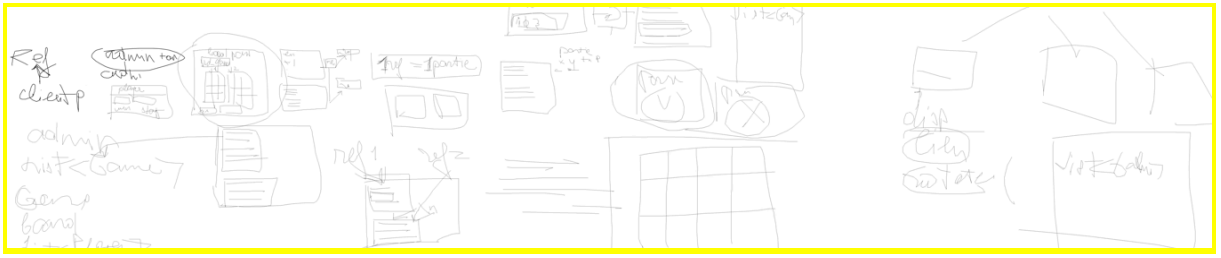

Image 6: Capture de l'écran de l'utilisation d'outil Paint pour modéliser un problème