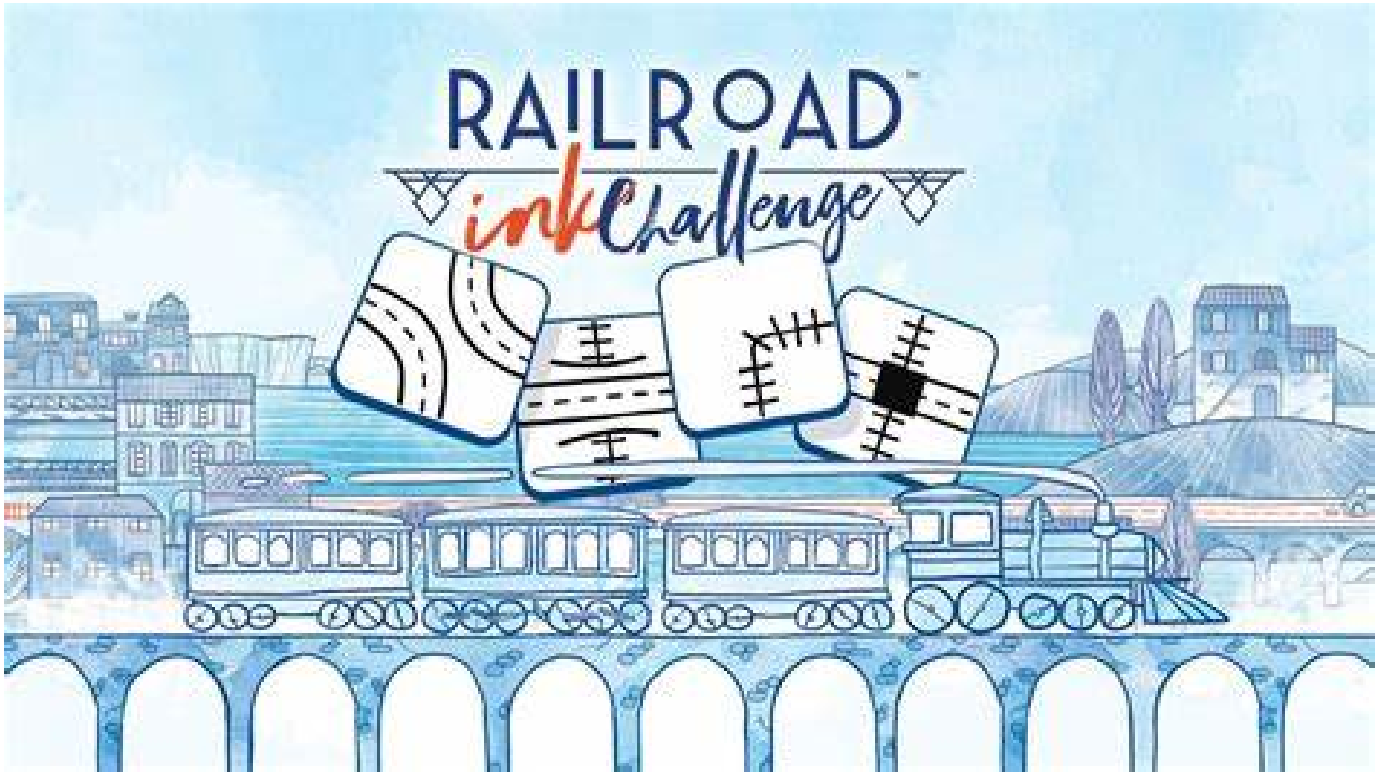
Dossier Rétrospectif



Réalisé par :

Agouni Douha
Boussaadane Hiba
Clabau Perrine
Vanelsland Marvin
Tartare Théophane

Université de Lille

I/ Présentation du projet

Ce projet consiste en la création d'un jeu de plateau en réseau, permettant à plusieurs joueurs de participer à distance avec une communication en temps réel via WebSocket. L'objectif principal est de concevoir une expérience interactive où les joueurs peuvent s'affronter en ligne, tout en garantissant une logique de jeu bien structurée et une communication fluide grâce à un serveur WebSocket. Ce projet met l'accent sur l'importance du travail collaboratif et de l'organisation en équipe pour réaliser un projet complet.
Dans le cadre de ce projet, nous avons implémenté une interface graphique, une librairie des messages autorisés dans le jeu, une librairie des règles du jeu à respecter, ainsi que plusieurs clients, tels que le client Arbitre, le client Joueur et le client Idiot.

Présentation des composantes importantes:

Composantes : essaie reflector début du projet, librairie règles du jeu, interface graphique, librairie des messages, les clients Arbitres, Joueurs et Idiots.

➔ **Composante : Reflector:**

- Nous avons implémenté deux **réflecteurs WebSocket**, l'un en **JavaScript** et l'autre en **Python** : ils établissent une connexion WebSocket avec un serveur sur `ws://localhost:3000`, lisent un fichier, et envoient son contenu ligne par ligne. Chaque message envoyé est affiché dans la console, et toute réponse reçue du serveur est également affichée.
- Les deux implémentations gèrent aussi les erreurs et la fermeture de connexion. En plus de leur rôle de relais de messages, ils maintiennent un **historique des messages** permettant ainsi une traçabilité des échanges entre les clients et le serveur.

➔ **Composante : Message:**

- Cette composante gère la communication WebSocket avec un serveur pour l'échange de messages structurés. Elle valide les messages selon des modèles prédéfinis, assurant que chaque commande respecte un format précis (ex: `ENTERS`, `THROWS`, `PLACES`).

- Les messages reçus sont validés avant d'être traités. Un système de gestion des erreurs et des événements assure la stabilité de la connexion.

➔ Composante : Régles du jeu

1. Cas d'utilisation:
   - Cette composante définit les règles du jeu permettant :
     - la gestion de tuiles ;
     - l'organisation des tours ;
     - calcul du score ;
     - la gestion des messages ;
     - la gestion du plateau de jeu ;

2. Packages et principales classes:
   - **regles.src** : contient l'implémentation de toutes les classes représentant les règles du jeu.
   - **regles.test** : contient les classes de test associées pour les classes précédentes.

3. Effort de développement :
   - Nombre de classes : 9 classes principales et 6 classes de test.
   - Nombre de tests :
     - **DesTest** : 2
     - **PlateauTest** : 8
     - **TestReglesDuJeu** : 5
     - **TestScoring** : 5
     - **TourTest** : 3
     - **TuileTest** : 7

   - Personnes impliquées :
     - Perrine Clabau
     - Hiba Boussaadane
     - Douha Agouni

➔ Composante : interface graphique

L’interface graphique représente l’interface par lequel le joueur va interagir avec le jeu. Entre autres, elle s'occupe d’afficher le plateau de jeu et permet au joueur de placer les tuiles sur le plateau grâce à sa souris. Le rôle de l’interface n’étant pas de communiquer directement via le réflecteur, les interactions avec celui-ci sont donc limitées à envoyer des messages au réflecteur pour annoncer le placement d’une Tuile.

Personnes impliquées :

- TARTARE Théophane
- Vanelsland Marvin

➔ **<u>Composante : client Joueur et Arbitre</u>**

Les clients Joueur et Arbitre vont mettre en commun les deux composantes présentées précédemment pour que le jeu fonctionne correctement. Déclinés en plusieurs versions selon le type de jeu voulu, les éléments qui sont communs à toutes ces versions sont que le client Joueur lance l’interface graphique, prévient de son entrée dans le réflecteur et attend que l’arbitre prenne en note le nouveau joueur pour décider de la marche à suivre. En somme, le client Joueur fait le lien entre l’interface graphique et le client Arbitre fait le lien entre le client Joueur et les règles.

Les versions implémentées et leurs règles sont :

- V1 : Prend en note les joueurs, leur affiche un plateau unique à chacun et tire les Tuile qui pourront être placées depuis l’interface graphique.

Personnes impliquées :

- TARTARE Théophane
- Vanelsland Marvin
- Perrine Clabau
- Hiba Boussaadane
- Douha Agouni

II/ Déroulement du projet :

a/ Organisation du travail :

Concernant notre organisation, nous avons divisé le travail en deux étapes : la phase de conception et la phase de développement.

Durant la phase de conception, nous avons commencé par analyser les règles du jeu et définir les fonctionnalités principales à implémenter. Cela inclut la modélisation des classes nécessaires (Joueur, Arbitre, Dés, etc.) et la définition des interactions entre elles. Nous avons également réfléchi à l'utilisation de la communication via WebSocket pour gérer les échanges en temps réel.
Durant la phase de développement, nous avons organisé la répartition du travail en formant des sous-groupes, chacun étant responsable du développement de différentes composantes du projet. Un sous-groupe a travaillé sur la logique de jeu et les règles, un autre s'est concentré sur la communication avec le WebSocket.
Cette approche nous permet de travailler en parallèle sur plusieurs aspects, d'optimiser notre efficacité et de mieux gérer le temps imparti. Ensuite, nous mettons en commun ces différentes composantes pour assurer la cohérence et le bon fonctionnement du projet.

Nous avons aussi mis en place une entraide active. Quand une personne rencontrait des difficultés ou était bloquée, nous prenions le temps de nous entraider, de partager nos connaissances et de réfléchir ensemble aux solutions possibles. Cela a non seulement renforcé notre esprit d'équipe, mais aussi permis à chacun d'apprendre davantage sur les parties du projet développées par les autres.

b/ Journal de bord

06/01 – 10/01 : Formation du groupe et prise en main du sujet. Développement du premier programme en Python permettant

d'envoyer une ligne dans un fichier texte au réflecteur et de lire le fichier.

13/01 – 17/01 : Ajout de dossiers pour structurer le projet. Début de la création de l'interface graphique et première implémentation de la classe "Tuile". Extraction des règles et informations importantes de l'énoncé pour mise à jour du fichier README.

20/01 – 24/01 : Réorganisation des dossiers du projet. Ajout d'un dictionnaire d'images et implémentation du placement des tuiles sur la grille. Ajout des flèches dans l'interface graphique. Développement de la classe "Plateau" et de la classe “Traces", ainsi que des classes "Scoring" et "ReglesDuJeu", accompagnées de leurs tests respectifs.

27/01 – 31/01 : Ajout de threads pour le client Idiot. Développement du code pour le placement des tuiles et commandes associées dans l'interface graphique. Amélioration de la classe "Plateau" et correction des tests de la classe "Scoring". Première implémentation de la classe "Message".

03/02 – 07/02 : Ajout du code pour le placement des tuiles dans la grille. Modification des classes pour la gestion des tuiles spéciales. Développement du stockage et de la gestion de plusieurs tuiles dans la grille, ainsi que la gestion du retournement des tuiles. Mise en place du drag and drop dans l'interface graphique. Envoi du placement des tuiles au réflecteur et ajout de la bibliothèque des messages. Modification des classes de règles et correction des tests de la classe "Plateau".

10/02 – 14/02 : Réorganisation des attributs pour faciliter la lecture du code. Réglages des détails sur les événements de souris (mouseEvent) dans l'interface graphique. Amélioration des classes "Plateau" et "PlateauTest". Correction des erreurs dans l'interface graphique et ajout de la gestion du réflecteur. Ajout de la gestion des rotations pour les tuiles spéciales. Début de l'implémentation de la classe "Tour".

17/02 – 21/02 : Gestion des rotations des tuiles spéciales dans l'interface graphique. Ajout de l'interaction avec ces tuiles spéciales et modification de l'affichage des tuiles spéciales.

24/02 – 28/02 : Réglages dans l'interface graphique pour la gestion des tuiles spéciales et des symboles de sortie. Modification des classes "Tuile" et "TuileTest" pour inclure les rotations et la symétrie des tuiles. Implémentation de la classe "Tour" et de ses tests, ainsi que du "ClientArbitreV1".

03/03 – 07/03 : Développement du main pour la gestion des règles du jeu. Modification de la classe "Message" pour inclure l'envoi et la réception via WebSocket. Ajustement de l'interface graphique pour un meilleur alignement des éléments et amélioration du placement des tuiles. Le code permet désormais de choisir entre tuiles normales et spéciales.

10/03 – 14/03 : Réorganisation du projet en ajoutant des interactions avec le réflecteur en JavaScript. Implémentation de la gestion des tuiles "dés" et gestion de plusieurs clients Idiot. Amélioration du "ClientArbitreV1", ajout du début de l'implémentation du "ClientJoueur" et création du dictionnaire de tuiles pour lier les tuiles à l'interface graphique. Modification des représentations des tuiles pour une meilleure cohérence avec l'interface graphique. Correction des classes et du "Main".

17/03 – 21/03 : Modification du stockage des tuiles dans l'interface graphique. Ajout du "ClientArbitreV1bis" et de l'intégration des joueurs dans le "ClientArbitre". Modification de ce dernier pour qu'il renvoie les bonnes tuiles, qu'elles soient spéciales ou non. Amélioration de l'avancement du "ClientJoueur" dans l'interface graphique. Ajout de la classe "JoueurTest" et modifications pour améliorer la documentation. Ajout de tests pour la classe "Des" et documentation des classes "Plateau", "ClientArbitreV1", et UML des règles. Élaboration du dossier rétrospectif.

c/ Répartition des tâches

- Clabau Perrine:

pythonWebSocket.py et texte.txt : Ce code Python utilise la bibliothèque websocket pour établir une connexion WebSocket avec un serveur situé à ws://localhost:3000. Ce code envoie les trois lignes du fichier texte.txt après les avoirs lu au serveur dès que la connexion est établie.

Des.py et DesTest.py: La classe Des implémenter un lancer de dés aléatoires. Chaque face d'un dé correspond à une tuile prédéfinie. Il y a 4 dés : 3 avec des tuiles normales et 1 avec des tuiles spéciales

Joueur.py et JoueurTest.py : Cette classe implémente un joueur du jeu qui utilise les dés et les tuiles. Ce joueur ne peut pas utiliser plus de 3 tuiles spéciales dans la partie(7 round). Gère l'ajout, la suppression et la gestion des tuiles spéciales.

Tour.py et TourTest.py : La classe Tour gère un tour de jeu dans lequel les joueurs tirent des tuiles et les placent sur leurs plateaux. Ces joueurs placent ces tuiles en indiquant une valeur x et y pour placer la tuile. Cette classe utilise la connexion WebSocket pour pouvoir envoyer les messages de placement des tuiles des joueurs.

Message.py : Cette classe Message représente la bibliothèque des messages acceptée dans ce jeu. Cette classe gère la communication avec un serveur via WebSocket. Elle se connecte au serveur, envoie des messages et reçoit des réponses tout en validant le format des messages.

Tuile.py et TuileTest.py : Cette classe implémente une représentation d'une tuile dans le jeu. Une tuile est représentée avec quatre côtés, chacun ayant une connexion de type énumération VIDE= 'E', ROUTE=

‘R’ ou RAIL = ‘T’. Une tuile peut être représentée avec une rotation ou une symétrie. De plus, elle permet de vérifier si deux tuiles peuvent être connectées entre elles sur un côté donné, en tenant compte de toutes les rotations et symétries possibles.

ClientJoueur.py : Cette classe représente un client Joueur qui joue donc place les tuiles et envoi un message BLÂME quand la tuile ne peut pas être placée car les voisins sont incompatibles. Malheureusement cette classe ne marche pas bien car j’ai modifié la classe interGraphic pour ajouter les messages BLÂME mais cela envoi un blâme que sur certaine tuile mal placé. Par la suite nous avons défini un autre client Joueur avec d’autre fonctionnalités car j’ai compris après que c'était le client Arbitre qui envoyait un message BLÂME au reflector et non le client Joueur.

DictionnaireTuile.py : Ce fichier impelmente un dictionnaire qui represente les tuiles possibles avec en clés le nom des tuiles (exemple ‘HH’ , ‘Sc’ …) et en valeur la représentation des tuiles avec leur 4 côtés (‘R’ pour route , ‘E’ pour vide et ‘T’ pour rail). Cela permet d’associer les tuiles à l'interface graphique.

Retour d’expérience personnel :

Ce projet a été une expérience intéressante et enrichissante. Cependant, j’ai eu du mal à le comprendre au début. Il m'a fallu beaucoup de temps pour assimiler les règles du jeu et comprendre l’implémentation. Malgré cette difficulté, le travail en équipe s’est très bien passé. Nous avons pu nous entraider et avancer ensemble, ce qui a rendu le projet plus agréable.

- Douha Agouni :

Plateau : Cette classe représente le plateau du jeu sur lequel les joueurs pourront placer leurs tuiles.
Elle commence par initialiser le plateau en créant une grille dont la taille est passée en paramètre. Elle contient également deux listes pour stocker les dés spéciaux et les tuiles à placer. De plus, elle définit des méthodes garantissant que les joueurs respectent bien les règles du jeu, par exemple en vérifiant si un emplacement est valide avant de placer une tuile et si celle-ci est correctement positionnée sur le plateau.

PlateauTest : J'ai dû modifier certaines classes que mes collègues avaient implémentées, telles que *Tuile* et *Dés*, afin de faire passer les tests des méthodes de la classe *PlateauTest*.

clientArbitreV1 : J'ai également implémenté la première version de *ClientArbitre*, qui est censée lancer l'interface graphique et effectuer le lancer de dés. Je l'avais d'abord développé en JavaScript, mais j'ai rencontré des difficultés pour l'intégrer au reste du code, car les autres classes étaient en Python. J'ai donc refait une version en Python, qui lance correctement les dés. De plus, j'ai ajouté quelques méthodes dans *InterGraphic* pour afficher les dés lancés par l'arbitre sous le plateau de jeu.
Néanmoins, j'ai rencontré un problème : la connexion se fermait après le lancer de dés par l'arbitre.

Retour d'expérience personnel :

- Le concept du jeu lui-même était intéressant, le travail en groupe s'est bien passé, nous avons réussi à répartir les tâches, sans rencontrer de conflits. Cependant, nous avons au début rencontré

quelques difficultés, notamment liées à l'utilisation de plusieurs langages de programmation, et faire fonctionner tout le code n'a pas été évident.

- Hiba Boussaadane:

Premiere implementation des classes Tour.py et Message.py: j'ai réalisé une premiere version des classes *Tour.py* et *Message.py* .

mainn.py et la correction des classes pour garantir son bon fonctionnement: Des ajustements ont été apportés aux classes existantes pour assurer la compatibilité et la fluidité de l'exécution du programme principal. Dans le mainn.py, le jeu est initialisé en suivant plusieurs étapes essentielles. Tout d'abord, il crée un ensemble de tuiles (normal et spéciales) qui seront utilisées pendant le jeu. Ensuite, il initialise le plateau de jeu et les dés avec les tuiles définies. Après cela, il crée deux joueurs et les ajoute aux règles du jeu. Le jeu commence ensuite avec l'appel de la méthode *commencer_partie()*. Le programme met en place une boucle où chaque joueur exécute son tour à tour de rôle. À chaque tour, les règles du jeu sont mises à jour et le round est terminé une fois tous les joueurs passés. La partie se termine lorsque la condition *est_partie_terminee()* est remplie, et le programme affiche alors un message indiquant la fin du jeu. Le tout est géré avec l'aide de WebSocket pour la communication entre les joueurs.

Scoring.py et TestScoring.py: La classe Scoring contient des méthodes statiques pour calculer le score d'un joueur : *calculer_reseaux_connectes* compte les cases connectées, *calculer_chemin_le_plus_long* trouve la longueur du chemin le plus long d'un type donné, *calculer_cases_centrales* attribue des points pour les cases centrales occupées, et *calculer_penalites* compte les cases incomplètes. La méthode *calculer_score_total* additionne ces valeurs et soustrait les pénalités pour obtenir le score final. La classe inclut aussi la méthode privée *_trouver_sorties_connectees* pour repérer les sorties

connectées. Les tests unitaires dans TestScoring.py vérifient le bon fonctionnement de ces méthodes.

ReglesDujeu.py et TestReglesDuJeu.py: La classe ReglesDujeu gère les règles du jeu en ajoutant des joueurs, en commençant et terminant les rounds, et en vérifiant si la partie est terminée. Elle gère les grilles de jeu, les scores et les tuiles spéciales des joueurs. La méthode *commencer_partie* initialise une nouvelle partie, tandis que *commencer_round* et *terminer_round* gèrent les rounds. *est_partie_terminee* vérifie si le nombre maximal de rounds est atteint, et *utiliser_tuile_speciale* permet aux joueurs d'utiliser une tuile spéciale. Des tests unitaires dans *TestReglesDuJeu.py* valident ces fonctionnalités.

UML des règles du jeu : j'ai réalisé aussi L'UML des règles du jeu présente la structure et les relations entre les classes dans le package regles, en mettant en évidence les méthodes essentielles pour la gestion des règles du jeu.

Retour d'expérience personnel :

Ce projet a été une expérience à la fois enrichissante et stimulante. Le travail en groupe a été très efficace, et la communication entre les membres de l'équipe s'est très bien déroulée. Travailler sur la mise en place des règles du jeu m'a permis de renforcer mes compétences en programmation orientée objet. Cependant, j'ai rencontré des difficultés lors de la réalisation du mainn.py, comme la gestion des erreurs lors du lancement des dés et l'intégration du WebSocket.

- Vanelsland Marvin :

Historique : Ce client se connecte à un serveur WebSocket (ws://localhost:3000), écoute les messages entrants et les enregistre dans historique.txt. Lorsqu'il se ferme, il vide le fichier.

interGraphique : Ce fichier implémente la classe Dessin qui représente l'interface graphique du jeu en PyQt5. Cette classe sert à gérer l'affichage du plateau et la manipulation des tuiles par l'utilisateur. On y retrouve le plateau de jeu, la tuile sélectionnée qui peut être tournée, ainsi que les différentes tuiles spéciales et celles données par les dés.

ClientIdiot : Ce client se connecte au serveur websocket et permet de lancer l'interface graphique sans la gestion des règles. Nous avons rencontré des difficultés pour démarrer l'interface graphique et établir une communication avec le réflecteur depuis le clientIdiot. Pour résoudre ce problème, nous avons utilisé des threads, ce qui permet de séparer la communication avec le réflecteur et l'exécution de l'interface graphique, évitant ainsi tout blocage.

clientArbitreV1bis : J'ai aussi fait une version du clientArbitre qui ouvre l'interface et lance les dés et les affiches. J'ai essayé de mettre en place le lancement des dés pour chaque joueur, mais je me suis orienté vers une mauvaise piste en cherchant à faire lancer les dés des joueurs par l'arbitre. Ce qui est impossible, le réflecteur bloque les commandes provenant d'un identifiant qui n'est pas le sien pour éviter l'usurpation d'identité .

Retour d'expérience personnel :

Le projet s'est bien déroulé dans l'ensemble, cependant, j'ai rencontré quelques difficultés à comprendre l'aspect "tournoi" du projet. Au cours du projet, je me suis davantage orienté vers l'aspect graphique, ce qui a été très intéressant, car c'est un point que nous avons assez peu abordé lors des projets précédents.

- TARTARE Théophane :

Dictionnaire.py :

est un fichier contenant plusieurs dictionnaires notamment un qui a pour clef le nom des tuiles et a pour valeur le lien vers l'image de la tuile ou encore un autre qui prend l'orientation par exemple N utilisé dans plusieurs fichiers mais principalement dans interGraphic.

Historique.js :

ce fichier est une conversion de Historique.py de Marvin en javascript pour pouvoir comprendre comment le réflecteur fonctionne réellement

Connexion.js :

ce fichier est une conversion de Conexion.py en javascript pour pouvoir comprendre comment le réflecteur fonctionne réellement

interGraphique.py :

Ce fichier implémente la classe Dessin qui représente l'interface graphique du jeu en PyQt5 .Cette classe sert à gérer l'affichage du plateau et la manipulation des tuiles par l'utilisateur.On y retrouve le plateau de jeu, la tuile sélectionnée qui peut être tournée, ainsi que les différentes tuiles spéciales et celles données par les dés.

plus précisément j'ai implanté la création du plateau (grille, chiffre, lettre, sortie,) l'affichage et le placement des tuiles spéciales et normal de la tuile sélectionnée avec leur orientation en meme temps que simplifier ou ameliorer les mouse event.

clientJoueurV1.py :

Ce fichier marche en tandem avec clientArbitreV1.py implémente un joueur quand ont l'ouvre il demande quelle ID ont veut pour jouer et un fois choisi nous ouvre le plateau de jeux et récupère les tuile que le client Arbitre envoie au réflecteur, les et les stock et les affiche dans l'interface graphique il envoie aussi au réflecteur quand une tuile est placée.

Arbitre.py :

une Class utilisée par les clientArbitre et implémente un Arbitre stockant les id des joueurs, leur main et vérifie s' il faut envoyer un blâme , tirer les dée etc.

Retour d’expérience personnel :

Le projet était très intéressant à réaliser, bien qu'au début, j'étais plutôt perdu quant à l'ordre des étapes et à la manière d'implémenter le projet. J’ai donc vécu une expérience différente des projets ou TP précédemment réalisés, où les étapes à suivre étaient soit implicites, soit clairement indiquées dans le sujet. Je me suis concentré sur l’interface graphique et les clients, ce qui est très satisfaisant, car chaque petit progrès est immédiatement visible.

Conclusion :

Ce projet nous a permis de renforcer nos compétences en programmation orientée objet, notamment dans la conception et la structuration d'un code collaboratif. Nous avons également découvert les aspects techniques liés aux communications réseau avec les WebSockets, ce qui nous a aidé à mieux comprendre les échanges de données en temps réel.

Travailler en groupe nous a appris à communiquer efficacement, à écouter les idées des autres et à partager nos connaissances. Surmonter ensemble les difficultés techniques nous a permis de réaliser l'importance de la collaboration et de l'entraide dans un projet complexe.

Ce projet nous a aussi permis de mieux gérer notre temps et de nous adapter aux imprévus. Au final, nous en ressortons plus confiants dans notre capacité à aborder des projets techniques ambitieux et à nous adapter à de nouveaux outils.