Romain BARRE **ING1 GI GROUPE 3**
Yann ETRILLARD
Thibault GARCIA-MEGEVAND
David KUSMIDER
Robin MENEUST

RAPPORT DE PROJET

CY PATH

Groupe 2

Julien MERCADAL

SOMMAIRE

INTRODUCTION

Pour ce projet de fin d'année, nous avons choisi de réaliser le programme CYPATH qui est une reproduction du jeu du Quoridor, à l'aide du langage de programmation Java.
Le Quoridor est un jeu de société stratégique dans lequel deux ou quatre joueurs s'affrontent pour atteindre le bord opposé du plateau de jeu. Pour rendre la tâche plus complexe, chaque joueur dispose d'un nombre de barrières qu'il peut placer pour bloquer le chemin de ses adversaires.

Nous avons sélectionné ce projet parmi tous puisque étant pour certains de grands joueurs de jeux de plateau, il a immédiatement attiré notre attention. D'autant que certaines personnes connaissaient déjà ce jeu, ce qui leur a permis tout de suite d'imaginer des manières de l'implémenter. La recherche de chemin avec un algorithme différent de ceux vu en cours nous a également beaucoup intéressée.

L'objectif de ce projet est de concevoir et d'implémenter notre propre version du jeu du Quoridor. Pour cela, nous utiliserons les principes de la programmation orientée objet offerts par Java et les nombreuses librairies chargées dynamiquement. Cet ensemble d'outils nous permettra de modéliser efficacement les différentes entités du jeu telles que les joueurs, les barrières et le plateau. Nous utiliserons également nos connaissances apprises en algorithmie procédurale avancée ce semestre pour la modélisation du plateau.

ORGANISATION

Construction de l’organisation

L’équipe que nous avons formée n’est pas nouvelle, à travers plusieurs projets, nous avons eu l'occasion d’améliorer notre fonctionnement afin d’obtenir des résultats satisfaisants. Nous connaissons chacun nos points forts ainsi que nos points faibles, ce qui a permis de distribuer les tâches de manière optimale.

Une fois la première réunion d’information passée, comme pour tout projet nécessitant un langage de programmation objet, nous avons d’abord commencé par réaliser les différents diagrammes nécessaires pour avoir une meilleure visualisation du projet dans son ensemble. Cette étape préliminaire nous a permis d’établir une liste exhaustive de tâches à compléter afin de concevoir CY Path. Chaque fonctionnalité majeure est divisée en plusieurs tâches comme vous pouvez le voir dans le fichier tasks.txt sur GitHub. Nous avons une date de départ, l’estimation de la date de fin, la date de fin réelle, et les problèmes rencontrés. Nous nous sommes ensuite répartis les tâches entre nous tout au long du projet. Dès qu’une personne avait fini sa tâche, il pouvait en prendre une nouvelle dans celles restantes dans tasks.txt, ou retravailler sur des tâches précédentes (pour les déboguer notamment).

Pour le “versionning” du projet, nous avons continué à utiliser des outils comme GitHub, ce qui facilita grandement notre organisation. Mais nous avons aussi suivi les indications de notre professeur référent, ainsi, nous avons une branche dev utilisée pour les versions en cours de développement et une branche master. La branche master correspondait à la dernière version stable de notre projet, avec le readme à jour notamment. Cette branche a été fusionnée chaque soir avec la branche dev. Avant chaque fusion, des tests de base ont été faits pour s’assurer que le programme fonctionnait. Par exemple, on vérifiait ce qu’il se passait lorsqu’un pion arrivait sur son côté gagnant ou encore lorsqu’un nom de fichier de sauvegarde, ou son contenu, était invalide. Mais cela n’a pas empêché les versions sur la branche master de contenir quelques bugs, passés entre les mailles du filet, qui ont alors été corrigés le lendemain.

Nous nous sommes tous réunis chaque jour depuis le début du projet pendant plusieurs heures sur Discord (en appel vocal) pour évaluer l'avancée du projet, poser des questions aux autres membres, et surtout s’entraider lorsque nous faisions face à un problème. Ainsi, à la fin de chaque journée, nous refaisons un point sur le projet pour déterminer quelles directions prendre par la suite afin de faciliter et d’optimiser notre implémentation.

Au début, nous avions prévu d'utiliser Ant pour faciliter les tâches répétitives comme la création d'archives exécutables ou de la génération de la documentation, mais une fois la partie JavaFX commencée, nous avons basculé sur Maven pour plus de simplicité, notamment au niveau des dépendances. Ce fut également l'occasion de découvrir un nouvel outil non vu en cours.

Problèmes organisationnels rencontrés

Un des premiers problèmes rencontrés fut la prise en main de Git pour plusieurs membres du groupe. Mais sur un projet de cette taille, cela nous a permis à terme de gagner un temps considérable. Nous avons notamment, à un certain point du projet, dû travailler tous sur un même fichier. C'est dans ces situations que Git a été particulièrement utile, en nous aidant à résoudre les conflits et fusionner plusieurs versions.

Planning

Vue globale

La première semaine nous avons codé les fonctionnalités de base de l'application en mode console puis graphique.
Puis, la deuxième semaine, nous avons ajouté les fonctionnalités manquantes et avons revu notre code pour réduire les erreurs et éviter la duplication, et enfin, nous avons travaillé sur le rapport et les diagrammes demandés.
Pour plus de détails vous pouvez consulter le fichier tasks.txt et lire le planning ci-dessous.

Semaine 1

Lundi 15/05 :

- Premier contact avec le projet et mise en place d'un premier diagramme de classe
- Création de toutes les classes apparaissant dans ce diagramme de classe
- Premier script d'automatisation avec Ant
- Création du graphe du plateau de jeu (représentation en mémoire choisie pour le plateau)
- Création des joueurs et création des barrières

- Création de la fonction pour jouer un seul tour de jeu : pose de barrières et déplacement des pions
- Première version de l'algorithme de recherche de chemin
- Ecriture de la fonction de recherche des déplacements valide pour un pion
- Affichage du plateau en mode console

Mardi 16/05 :

- Changement de la matrice d'adjacence en liste d'adjacence
- Fin de création des barrières
- Séquence de jeu : condition de victoire et nombre de joueurs
- Création de sauvegarde
- Amélioration de la recherche de déplacements valides pour un pion

Mercredi 17 :

- Ajout du cas d'intersection de barrière
- Passage de Ant à Maven au projet avec notamment la génération d'un .jar contenant les dépendances (JavaFX, …)
- Ajout d'un script en YAML sur github pour générer les archives .jar du livrable (dans l'onglet Actions)

Jeudi 18 :

- Création de la grille du plateau de jeu avec JavaFX
- Ajout de la boucle de jeu pour le mode graphique
- Changement de l'affichage des coordonnées en mode console pour simplifier la pose de barrière
- Affichage des déplacements valides des pions dans le mode graphique et affichage de la case sélectionnée
- Ajout bouton de choix d'action pour le mode graphique
- Ajout de l'action "déplacer un pion" en mode graphique
- Affichage des barrières sélectionnées en mode graphique

Vendredi 19 :

- Modification de la structure du projets (des classes) pour séparer la partie graphique de la partie console pour que le joueur puisse lancer l'un de ces deux modes d'affichage au lancement du programme

Samedi 20 :

- Finalisation de la séparation du mode console et graphique
- Ajout d'un pion pour chaque joueur
- Ajout de l'action de placement de barrière
- Ajout d'un compteur de nombre de barrières restantes
- Ajout du système de sauvegarde des parties

- Ajout d'un bouton de sauvegarde
- Ajout de menus

Dimanche 21 :
- Séparation du projet en trois packages : abstraction, presentation et control
- Finalisation du système de sauvegarde pour le mode graphique

Semaine 2

Lundi 22:
- Ajout du chargement de sauvegardes
- Ajout d'un menu de fin de partie en cas de victoire

Mardi 23 :
- Demande à l'utilisateur de relancer la partie ou de fermer le programme au lieu de forcer sa fermeture en cas de victoire
- Amélioration du style des menus
- Ajout de la possibilité de passer son tour lorsque le joueur ne peut pas bouger, afin d'éviter une situation de blocage à quatre joueurs

Mercredi 24 :
- Ajout de la possibilité de changer les règles du jeu (taille des barrières…) dans le code directement

Jeudi 25 :
- Amélioration de la recherche de chemin
- Changement de la gestion des threads pour réduire l'utilisation du CPU en cas de création de nouvelles parties

Vendredi 26 :
- Création des diagrammes de séquences avec PlantUML
- Mise à jour des diagrammes de classe et de cas d'utilisation

Samedi 27 :
- Finalisation du rapport

DIAGRAMMES

Diagramme des cas d'utilisation

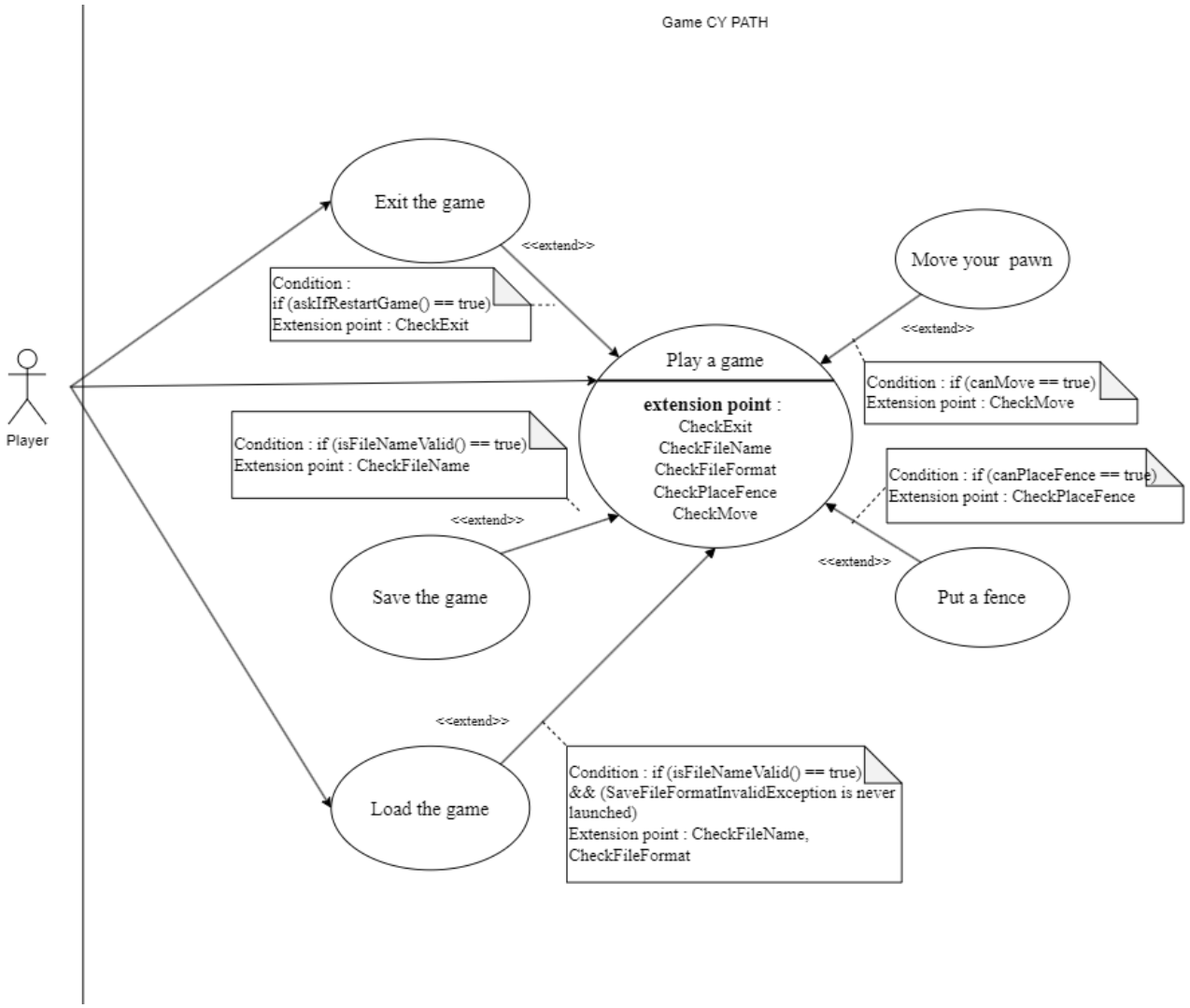
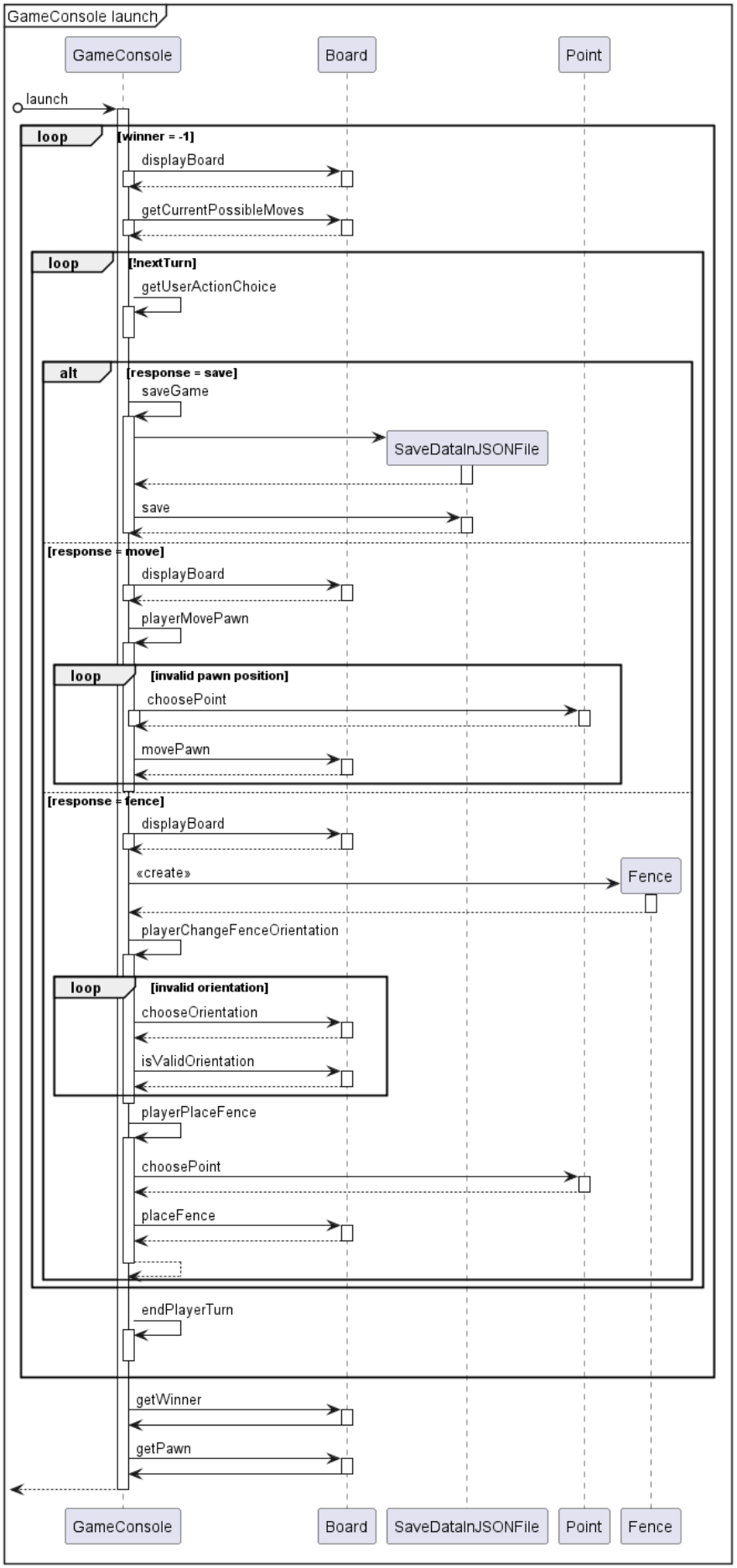
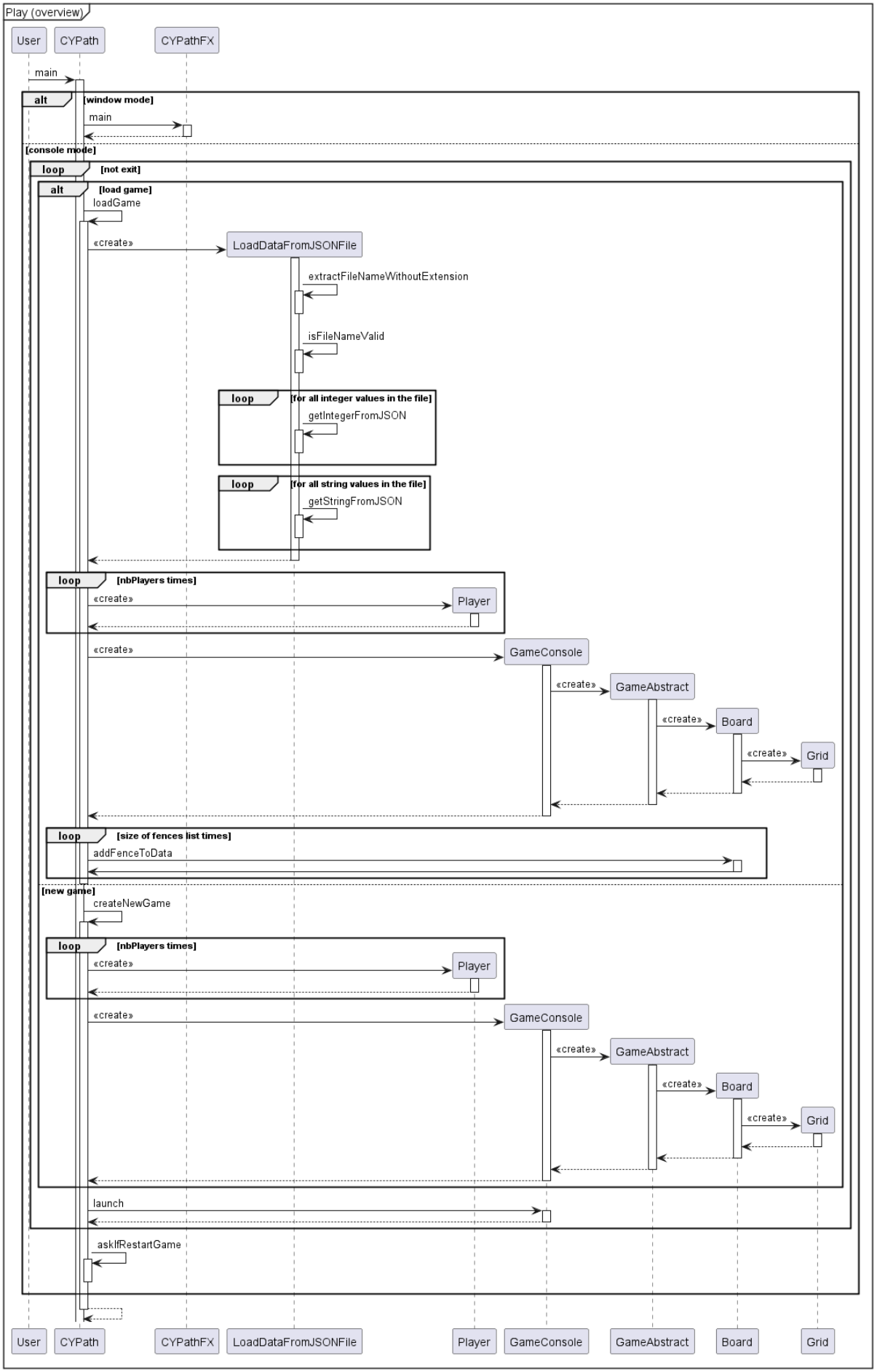
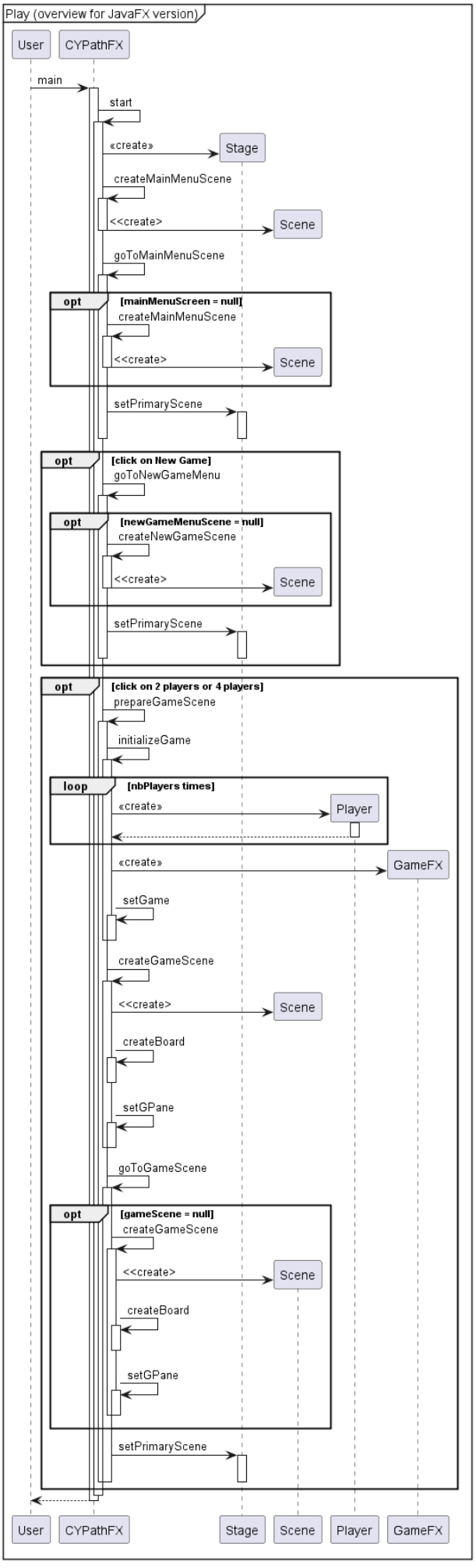


Diagramme des classes

Diagrammes de séquence



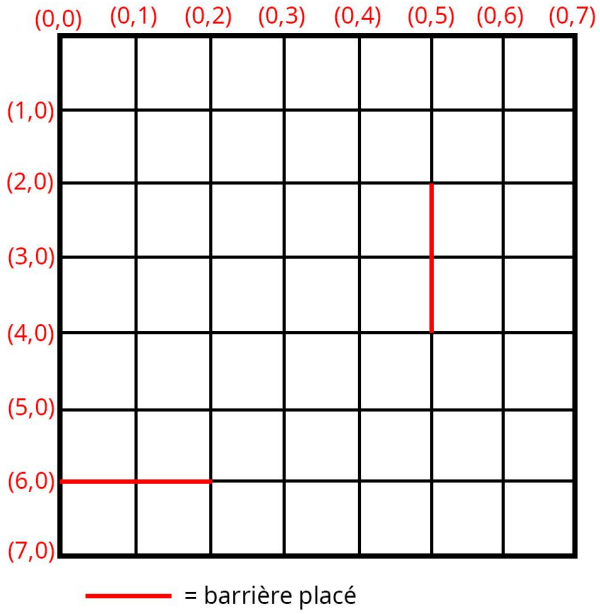

Play (overview)
User
CYPath
CYPathFX
main
alt
[window mode]
main
[console mode]
loop
[not exit]
alt
[load game]
loadGame
«create»
LoadDataFromJSONFile
extractFileNameWithoutExtension
isFileNameValid
loop
[for all integer values in the file]
getIntegerFromJSON
loop
[for all string values in the file]
getStringFromJSON
loop
[nbPlayers times]
«create»
Player
«create»
GameConsole
«create»
GameAbstract
«create»
Board
«create»
Grid
loop
[size of fences list times]
addFenceToData
[new game]
createNewGame
loop
[nbPlayers times]
«create»
Player
«create»
GameConsole
«create»
GameAbstract
«create»
Board
«create»
Grid
launch
askIfRestartGame
User
CYPath
CYPathFX
LoadDataFromJSONFile
Player
GameConsole
GameAbstract
Board
Grid

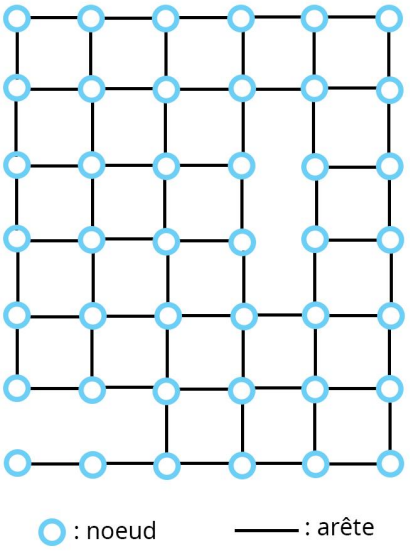

Play (overview for JavaFX version)
User
CYPathFX
main
start
«create»
Stage
createMainMenuScene
<<create>
Scene
goToMainMenuScene
opt
[mainMenuScreen = null]
createMainMenuScene
<<create>
Scene
setPrimaryScene
opt
[click on New Game]
goToNewGameMenu
opt
[newGameMenuScene = null]
createNewGameScene
<<create>
Scene
setPrimaryScene
opt
[click on 2 players or 4 players]
prepareGameScene
initializeGame
loop
[nbPlayers times]
«create»
Player
«create»
GameFX
setGame
createGameScene
<<create>
Scene
createBoard
setGPane
goToGameScene
opt
[gameScene = null]
createGameScene
<<create>
Scene
createBoard
setGPane
setPrimaryScene
User
CYPathFX
Stage
Scene
Player
GameFX

DÉVELOPPEMENT

Représentation du plateau

Nous nous sommes questionnés au début du projet sur la représentation des données du plateau dans notre programme.
Nous avons après réflexion, et en tenant en compte du fait qu'une recherche de chemins devait être faite après, décidé d'utiliser un graphe non orienté. Dans ce graphe les sommets sont des cases (des points avec des coordonnées entières) et les arêtes représentent l'existence d'un chemin entre les deux cases. Ainsi en ajoutant une barrière on supprime des arêtes de la façon suivante :

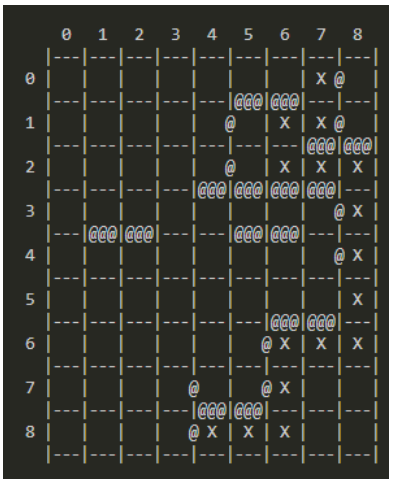




Nous utilisions alors dans nos premières versions une matrice d'adjacence de taille 81x81. Ce qui n'était pas du tout adapté au problème ici. En effet, chaque case du plateau est reliée à au plus 4 autres cases, alors qu'ici on considère que chaque nœud du graphe peut avoir jusqu'à 81 voisins. Ainsi nous avons changé cela en considérant une HashMap dont les clés sont des points dont les coordonnées correspondent à une case sur le plateau, et les valeurs, des listes d'adjacences correspondant aux cases voisines de la case à ces coordonnées. Pour la liste d'adjacence nous avons choisi un HashSet contenant des les coordonnées des voisins, puisqu'un voisin ne doit apparaître qu'une seule fois dans la liste d'adjacence.

Les barrières sont aussi stockées dans une liste dans la classe Board pour faciliter leur sauvegarde, en évitant notamment de refaire un parcours complet de la grille pour la sauvegarde d'une partie.

Pour vérifier l'existence d'un chemin entre chaque joueur et leur côté gagnant, nous avons utilisé l'algorithme "A étoile". Il existe une grande variété d'algorithmes de recherche de chemin, avec chacun plusieurs implémentations. Et A étoile faisait partie de ceux qui correspondaient le plus à la situation que nous avons ici. En effet, nous connaissons les coordonnées du point de départ et du point d'arrivée (sur le côté gagnant du joueur). Il est donc plus intéressant de nous diriger vers lui plutôt que de partir sur une recherche "aveugle" comme Dijkstra, puisque nous pouvons avoir une estimation de la distance entre un point étudié et la destination. Il est donc plus intéressant ici de prendre les points ayant une distance estimée à la destination qui est la plus petite possible. Cela se fait avec une file de priorité dans laquelle on met le point de départ puis tant que cette liste contient des noeuds à traiter ont enlève le premier (celui qui a la plus faible distance estimée) et on ajoute ses voisins à la file si ceux-ci n'ont pas déjà été rencontrés dans un chemin plus court entre eux et le départ.



Ici le coût (la distance estimée) est calculé en calculant la distance entre le point étudié et le point le plus proche du côté gagnant, on ne calcule donc que la valeur absolue de la différence entre des coordonnées y ou des coordonnées x selon les côtés considérés.

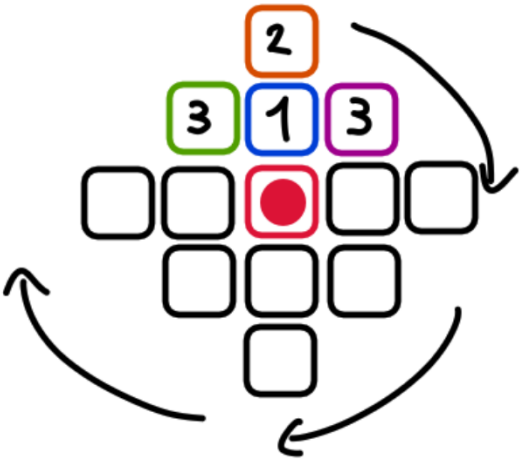
Dans une version précédente, on utilisait la norme de Manhattan pour calculer la distance entre le point de départ et le point au centre du côté gagnant, mais on parcourt alors plus de nœuds que nécessaires, d'où le changement.

Concernant la complexité de cet algorithme, nous avons ici dans le pire des cas une complexité polynomiale (si on parcourt une et une seule fois chaque sommet, hormis ceux du côté gagnant). Mais ici nous avons une grille avec 81 cases donc cette implémentation

reste largement suffisante, bien qu'elle puisse encore être améliorée, notamment en exploitant le fait que nous avons ici une grille.

Gestion des mouvements de pion

Lorsqu'un joueur souhaite se mouvoir, nous affichons sur l'interface graphique toutes les cases atteignables par ce dernier. C'est en cliquant sur l'une de ces cases que le joueur se déplacera. Pour cela, nous avons besoin d'une liste de toutes ces cases obtenue grâce à la fonction listPossibleMove. Nous avons remarqué que dans chaque direction les cases atteignables sous certaines conditions forment un T inversé de 4 cases. Et nous devons vérifier les cases dans l'ordre indiqué dans le schéma ci-dessous, de la manière suivante : si la case 1 est libre on ne retourne que celle-ci, si la 1 n'est pas disponible (à cause d'un pion ou d'une barrière), on vérifie la 2 et enfin si la 1 et 2 ne sont pas valides on regarde les deux cases 3 et on retourne celles libres. Nous avons donc choisi de faire fonctionner cette fonction comme un sonar et de faire tourner le "T" de vérification vers la droite par rapport à la position du joueur dans chacune des 4 directions. Dans une version précédente du programme, notre fonction était plus longue car on n'utilisait pas de matrice de rotation dans une boucle, mais on écrivait à la main les 4 groupes de cases à vérifier, ce qui donnait une fonction avec du code qui se répétait beaucoup.



Placement des barrières

Lorsqu'un joueur souhaite placer une barrière, nous devons vérifier trois conditions. Dans un premier temps nous devons vérifier si la barrière est sur le plateau. Dans un second temps nous devons vérifier si l'emplacement ne rentre pas en collision avec d'autres barrières. Pour cela nous parcourons la liste des barrières placée sur le plateau pour vérifier s'il y a une

collision avec l'une d'entre elles. La troisième condition doit vérifier si, malgré le placement de la barrière, les joueurs peuvent encore atteindre leur objectif. Pour cela nous ajoutons la barrière temporairement au graphe et nous vérifions s'il existe un chemin entre les pions et leur objectif. Cet ajout se traduit par le retrait des arêtes liant les nœuds entre eux, là où la barrière est posée, les nœuds représentant les cases. Si toutes ces conditions sont validées, on rompt les arêtes pour de bon.

Sauvegarde et chargement des parties

A propos de la sauvegarde des données d'une partie, nous avons opté pour le format de fichier JSON (JavaScript Object Notation). Nous avions déjà eu l'occasion dans de précédents projets de manipuler ce format. Sa syntaxe simplifiée qui se base sur un ensemble de paires clé-valeur nous permet d'écrire et de lire des données plus facilement que dans d'autres formats de données, comme par exemple XML.

Nous enregistrons les fichiers de sauvegarde dans un dossier par défaut ayant comme chemin "./data/saves".

Lorsque l'utilisateur souhaite sauvegarder sa partie, nous réalisons une vérification du nom du fichier afin que celui-ci soit valide, il doit contenir uniquement des caractères alphanumériques et certains caractères non alphanumériques. Si celui-ci est correct, nous lui ajoutons l'extension ".json".
De plus, nous contrôlons que le nom de fichier n'est pas déjà présent dans le dossier de sauvegarde dans quel cas, nous demandons confirmation au joueur s'il souhaite remplacer le fichier.

Nous enregistrons chacune des données nécessaires pour régénérer la partie en cours dans une variable de type JSONObject. A l'aide du principe de clé-valeur nous pouvons enregistrer nos données de type primitif dans celle-ci. Cependant pour enregistrer la liste des barrières ainsi que celles des pions, nous passons par une étape intermédiaire qui consiste à créer des variables de type JSONArray, contenu dans la variable précédente. Ces tableaux contiennent eux-même des variables de type JSONObject pour stocker leurs différents éléments respectifs.

Lorsqu'une personne souhaite charger une partie, nous vérifions que le nom du fichier de sauvegarde soit valide en utilisant les mêmes règles que précédemment.
Pour se prémunir de modifications erronées, nous effectuons une vérification de chaque type de données des variables stockées dans le fichier. De plus, nous vérifions que leurs

valeurs soient cohérentes selon des conditions que nous avons définies à l'intérieur de méthodes.

Dans le cas où toutes les valeurs sont cohérentes, nous pouvons ainsi reconstruire nos listes de barrières et de pions qui, avec le reste des éléments du fichier de sauvegarde, permettent de recréer une partie de jeu.

Tout au long de ce processus de sauvegarde et de chargement de parties, différentes exceptions peuvent être levées en fonction des erreurs repérées, telles que FileNameException si le nom du fichier est incorrect, FileAlreadyExistsException si le nom du fichier saisi par le joueur est déjà présent dans le dossier de sauvegarde, FileNotFoundException si le fichier spécifié par l'utilisateur n'a pas été trouvé, ParseException si une erreur apparaît durant le processus d'extraction des informations contenues dans le fichier de sauvegarde et SaveFileFormatInvalidException si les données du fichier de sauvegarde sont incorrectes.

Interface graphique avec JavaFX

Lorsque nous sommes en pleine partie l'interface JavaFX doit pouvoir évoluer en même temps que la partie. Pour ce faire nous avons fonctionné avec un principe de multithreading. C'est-à-dire que nous avons en réalité deux threads qui communiquent entre eux. Le premier est la partie JavaFX gérant la partie graphique et le second gère l'évolution de la partie et contient ses informations.

Nous avions initialement un Objet Thread créé à chaque création de partie, mais malgré plusieurs tentatives nous n'arrivions pas à l'arrêter, ce qui entraînait une augmentation importante de l'utilisation du CPU lorsque nous créons en boucle des nouvelles parties. Nous sommes alors passés aux Service et au Task, proposées par JavaFX, et lorsque une nouvelle partie est créée on ne crée plus de nouveau thread on ne fait que relancer le service associé à une partie, en lui redonnant un nouvel objet GameFX avec la méthode *restart* de Service.

À chaque tour, l'utilisateur peut faire deux actions principales : bouger son pion ou placer une barrière. Lorsqu'une de ces deux actions est réalisée, le booléen "fin de tour" devient vrai, alors le thread qui était en attente, lance le prochain tour. À chaque début de tour, nous avons souhaité que le joueur soit sur l'action "move". Pour cela la partie graphique qui était elle aussi en attente, clique artificiellement sur le bouton action pour le mettre sur "move" et mettre à jour les données visuelles.

Pour créer le plateau de jeu, nous avons considéré un GridPane de dimension 19 x 19, où nous avons une ligne sur deux pour les bordures des cases et une ligne sur deux pour les cases, de même pour les colonnes. D'où la dimension : 9 cases et 10 lignes de bordures. Pour passer des coordonnées de jeu au GridPane il suffit de multiplier par 2 pour les barrières et d'ajouter 1 si on considère une case.

Chaque case de ce GridPane est un StackPane. Cela nous permet de placer le fond des cases (un Rectangle) et un pion par dessus si besoin (un Circle).

Pour le placement des barrières et la coloration de celles sélectionnées avant ajout, nous utilisons une liste contenant les Line correspondant à ces barrières dans le GridPane, afin de les décolorer en cas de changement d'action, de placement de barrière ou de sélection d'une autre barrière.

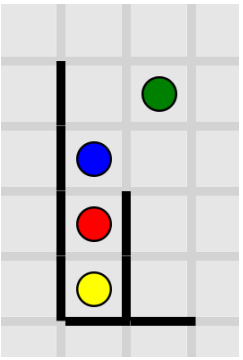
PROBLÈMES RENCONTRÉS

Classe trop longue

Nous étions loin d'imaginer le nombre de lignes de code et de classes que nous allions atteindre au final. Nous nous sommes donc rapidement retrouvés avec des fichiers de classes contenant un très grand nombre de lignes, ce qui a posé des problèmes évidents d'organisation et de lisibilité. Nous avons donc dû, et cela plusieurs fois au cours du projet, diminuer, réorganiser et créer de nouvelle classes afin d'y voir plus clair et faciliter la suite du développement.

Le cas match nul à 4 joueurs

Après avoir développé les prémices de l'interface graphique nous avons pu tester plus simplement notre code pour s'assurer qu'il prenait en compte tous les cas de figures. C'est alors que nous sommes tombés sur un problème, non pas technique, mais de compréhension. Que se passe-t-il quand un joueur ne peut pas se mouvoir ? C'est un cas de figure que l'on a rencontré dans la situation suivante :



Ici le joueur jaune ne peut pas jouer. Cette situation n'étant pas décrite dans les règles nous avons donc dû trouver nous même une issue à cette situation. Et nous avons fait le choix de proposer aux joueurs de pouvoir passer leur tour. Cette ajout de règle nous a alors demandé de rajouter un bouton "skip", permettant de passer son tour, apparaissant uniquement dans cette situation.

La séparation de l'affichage graphique et de la boucle de jeu

Lorsque nous lancions la boucle de jeu dans la classe CYPathFX (classe utilisée pour l'affichage graphique), la fenêtre de l'application se figeait. Cela est dû au fait que la partie graphique attendait la fin de la boucle de jeu pour continuer ce qu'elle faisait. Or la boucle

de jeu ne s’arrêtait jamais puisqu’elle attendait une entrée qui ne pouvait jamais lui être donnée (étant donné que la fenêtre ne répondait plus).
Il a donc fallu changer notre façon de relier les deux. Nous avons alors créé un Thread séparé pour la boucle de jeu (dans GameFX). Mais cela nous a amené un autre problème quelques versions plus tard : à chaque création de nouvelle partie, un nouvel objet Thread était créé, et l’ancien n’était pas arrêté, ce qui entraîna une augmentation importante de l’utilisation du CPU. Nous avons donc décidé d’utiliser une autre méthode : les classes Service et Task de JavaFX. Avec cette solution le nombre de threads n'augmente plus à la création de nouvelles parties et donc l’utilisation du CPU ne monte plus.

AXES D'AMÉLIORATION

Nous sommes fiers de ce que nous avons produit en respectant les contraintes de délais, cependant notre solution n'en reste pas moins perfectible, et nous avions dès le début du projet plusieurs fonctionnalités non demandées qui nous paraissaient intéressantes à implémenter. Par exemple, la sélection des barrières pourrait être améliorée. En effet, ici il faut survoler une case pour pouvoir placer la barrière, mais il serait peut-être plus simple (pour l'utilisateur) de survoler directement la bordure concernée. Un autre point que nous pouvions améliorer était la partie graphique, en effet, CY Path possède le minimum pour que l'expérience utilisateur soit agréable mais nous pouvions par exemple ajouter des animations pour le gagnant et mieux réaliser notre direction artistique.

Concernant les fonctionnalités bonus, il y en avait deux qui nous tenaient à cœur : le mode multijoueur en ligne et le mode joueur contre machine. Le premier pouvait rajouter la dimension réseau et les requêtes client-serveur, ce qui était particulièrement intéressant et surtout plus pratique que de jouer à quatre sur une même machine. Et pour challenger les utilisateurs, il pourrait être intéressant d'ajouter une intelligence artificielle contre laquelle jouer. Il y aurait plusieurs niveaux de difficultés.
Il serait aussi possible d'enregistrer les joueurs, alors associés à un compte, dans une base de données stockant par exemple leur nombre de victoires.

CONCLUSION

Ce projet fut une agréable expérience en équipe. Il nous a permis de développer nos compétences transversales telles que la communication et l'entraide. Nous avons également élargi notre socle de compétences en Java. Nous avons pu nous rendre compte de l'importance de réaliser un modèle de données détaillé dès le départ du projet, afin de structurer plus facilement celui-ci. En effet, nous avons souvent été confrontés à des problématiques techniques, des duplications de codes, ce qui a impliqué des modifications significatives lors de nos semaines de projet. Ce projet nous a également permis de nous familiariser avec des outils de développement tels que GitHub et Maven. Pour GitHub, nous avons notamment utilisé l'onglet Actions pour générer les archives exécutables et la documentation à chaque publication sur la branche master.