Université de Cergy-Pontoise

RAPPORT

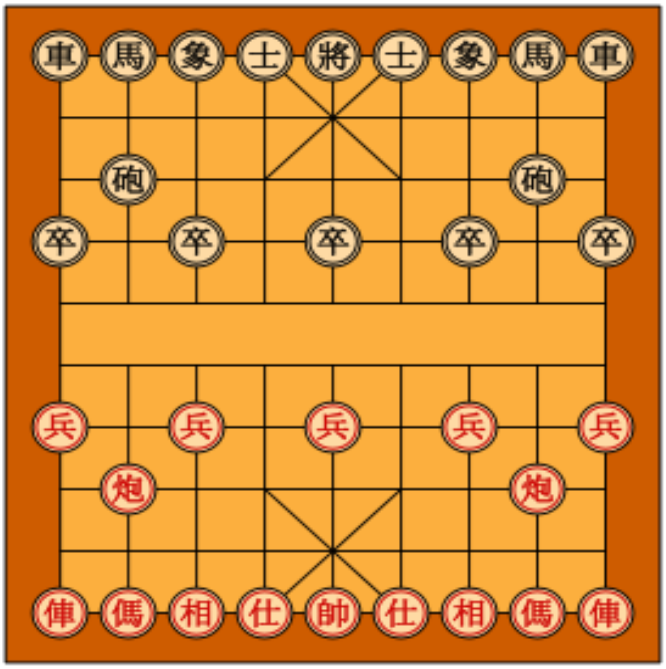
pour le projet Génie Logiciel
Licence d'Informatique deuxième année

sur le sujet

CHESS

rédigé par

Dorian CHENET, Almamy CAMARA, Alexis CHOLLET



Mai 2015

Table des matières

Table des figures

Liste des tableaux

Remerciements

Remerciements à l'équipe éducative : M. Liu pour sa conduite du module GLP et les TD de POO du premier semestre. Nos remerciements aussi à M. Lemaire qui a animé les cours magistraux de POO du semestre 3. Enfin, merci à nos camarades pour avoir été supportifs tout au long de l'année.

1 Introduction

Nous avons choisi le projet de GLP intitulé CHESS qui consiste à réaliser en Java un jeu d'échecs chinois appelé XiangQi. Ce projet a été un de nos premier choix lors de la phase d'attribution des sujets car il possédait tout les aspects techniques que nous recherchions : Alexis CHOLLET et Almamy CAMARA recherchaient du développement graphique tandis que Dorian CHENET recherchait plus un aspect algorithmique. Ce sujet était aussi intéressant car il permettait de pouvoir mettre en oeuvre des mécanismes poussés de Java (patterns et fonctionnalités entre-autres).

1.1 XiangQi, présentation

Le XiangQi est un jeu de plateau qui joue à deux, chaque joueur dispose de 16 pièces : 1 général, 2 conseillers, 2 cheveaux, 2 canons, 2 chariots et 5 soldats. Le but du jeu est le même qu'aux échecs occidentaux, c'est à dire mettre en échec le Roi (appelé général au XiangQi). Cependant, ce jeu possède un certain nombre de différences par rapport à la version occidentale des échecs, entre-autre :

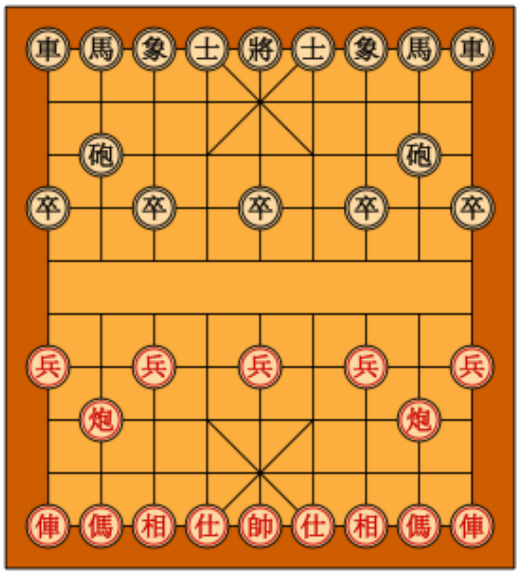


FIGURE 1 – Plateau du XiangQi

FIGURE 2 – Plateau des échecs occidentalles

— Les cases sont matérialisées par des intersections de lignes plutôt que des carrés de couleur.
— Dimensions : 10x9 emplacement (contre 8x8 pour le jeu d'occident).
— Le plateau du XiangQi possède des éléments d'environnement : la rivière, les palais.
— Le XiangQi possède des pièces que le jeu d'occident n'a pas : cannons et conseillers par exemple, nous y reviendrons.
— Le cavalier du XiangQi ne peut pas sauter par dessus une autre pièce.

Les pièces et le plateau seront expliqués plus en détail par la suite.

1.2 Projet, introduction

Notre projet a pour but de réaliser une version "occidentalisée" du XiangQi. Nous avons apporté un certain nombre de modifications au jeu, notamment un plateau plus grand, des pièces avec des mouvements différents ou encore des éléments d'environnement différents. Nous proposons aussi la possibilité de jouer non seulement joueur contre joueur mais aussi joueur contre IA (que nous avons appelé Chesster). Cette IA est un algorithme qui est capable d'évaluer et d'effectuer le meilleur coup possible (dans la mesure de nos capacités et nos délais de développement). Nous avons aussi, bien entendu, réalisé une interface graphique pour pouvoir jouer en temps réel.

Tous ces points sont repris dans le cahier des charges qui suit.

2 Cahier des charges

Avant toute chose, il est évident que tous les éléments cités ici sont pris en charge par le moteur du jeu que nous avons développé. Pour proposer une version occidentalisée du XiangQi, nous avons agi sur plusieurs points. Tout d'abord, voici les spécificités sur XiangQi :

2.1 Les XiangQi traditionnel

2.1.1 Le Plateau

Le plateau du XiangQi possède des dimensions qui lui sont propres : 10 positions de haut pour 9 positions de large. Chaque position est matérialisée par l'intersection de deux lignes (contrairement au plateau occidental qui possède des cases). De plus, ce plateau possède des éléments d'environnement :

- La rivière (zone vide au centre du plateau).
- Les palais, un palais est un carré de 3x3 positions, il est facilement repérable sur le plateau car il est croisé de deux diagonales. Les palais sont au nombre de 2, un pour chaque camp (rouge et noir).

Ces éléments d'environnement sont important car certaines pièces ont des déplacements qui sont dépendants de ces éléments.

2.1.2 Les Pièces

Le Général A l'instar des échecs occidentaux, chaque joueur possède un Général qu'il se doit de protéger. Le Général peut se déplacer d'une case horizontalement et verticalement à l'image du Roi des échecs occidentaux. La différence avec ce dernier est qu'il n'a pas le droit de sortir du palais dans lequel il est posé. De plus, les deux généraux rouges et noirs n'ont pas le droit de se retrouver directement face à face (sur la même ligne/colonne). Ils doivent donc toujours être sur une ligne/colonne différente ou alors être séparés par une pièce (de couleur quelconque) qui se trouverait entre les deux généraux.

Les Conseillers Chaque joueur possède 2 conseillers, ils sont initialement placés aux côtés du Général. Les Conseillers peuvent se déplacer d'une case sur les deux diagonales. Comme le Général, ils n'ont pas la possibilité de se déplacer en dehors du palais dans lequel ils sont initialement posés.

Les Elephants Chaque joueur possède 2 Elephants, ils sont initialement placés aux côtés des Conseillers. Les Elephants ne peuvent se déplacer que de 2 cases en diagonal et ne peuvent pas sauter par dessus une autre pièce. De plus, ils ne peuvent pas traverser la rivière, chaque Elephant possède donc 7 positions possibles sur le plateau. Leur rôle est principalement offensif.

Les Chevaux Chaque joueur possède 2 Chevaux, ils sont initialement placés aux côtés des Elephants. Les Chevaux se déplacent en "L" comme les cavaliers des échecs occidentaux mais un peu différemment. En effet, les Chevaux du XiangQi se déplacent d'abord d'une case en diagonal, puis d'une case horizontalement ou verticalement. Pendant leur déplacement, les Chevaux ne peuvent pas sauter par dessus une autre pièce, le joueur doit donc s'assurer que le chemin est libre avant de jouer.

Les Chariots Chaque joueur possède 2 Chariots, ils sont initialement placés aux extrêmités du plateau, aux côtés des Chevaux. Les Chariots sont l'équivalent des tours aux échecs occidentaux ; ils peuvent se déplacer d'autant de cases que souhaité verticalement et horizontalement. De plus, ils ne peuvent pas sauter par dessus d'autres pièces.

Les Canons Chaque joueur possède 2 Canons, ils sont placés deux lignes en avant dans le camp de chaque joueur. Un Canon peut se déplacer d'autant de cases que voulu verticalement et horizontalement mais il ne peut pas prendre de pièces ennemies. Pour pouvoir manger une pièce ennemie, les Canons doivent d'abord obligatoirement sauter par dessus une et une seule pièce. Ces pièces sont très offensives.

Les Soldats Chaque joueur possède 5 Soldats, ils sont placés près de la rivière, dans la partie centrale du plateau. Un Soldat ne peut initialement se déplacer que d'une case vers l'avant. Cependant, lorsqu'un Soldat traverse la rivière, il peut de déplacer d'une case en avant, vers la gauche ou vers la droite (mais jamais revenir en arrière).

2.2 Le XiangQi occidentalisé

Notre projet avait pour but de proposer une version "occidentalisée" du XiangQi, nous avons donc apporté un certain nombre de modifications au jeu traditionnel :

2.2.1 Le Plateau

- Nous avons agrandi le plateau, originellement de dimensions 10x9, il est désormais de dimensions 12x11.
- La rivière est désormais représentée par une simple ligne et les palais ont été agrandis (3x3 to 5x4).
- Les pièces ne sont plus placées sur des intersections de lignes mais dans des cases à la manière des échecs occidentaux.
- Nous avons ajouté différents éléments d'environnement, différents types de sol sur lesquels certaines pièces ont des déplacements spécifiques.
- Le joueur a le choix dans le menu de démarrage de choisir le plateau sur lequel il désire jouer parmi un des 4 plateaux, chacun comportant un environnement spécifique.

2.2.2 Les Pièces

- Les Elephants peuvent désormais se déplacer d'autant de cases que voulu dans les deux diagonales à la manière d'un fou aux échecs occidentaux.
- Certaines pièces auront des déplacements spécifiques en fonction du type de sol sur lequel la pièce se trouve.

2.3 Intelligence

Chesster Nous avons dû réaliser une intelligence artificielle capable de jouer en autonomie contre un humain. Dans le menu d'accueil, avant de lancer la partie, le joueur peut choisir de jouer soit contre un joueur humain, soit contre l'ordinateur. Cette intelligence nommée Chesster évalue les meilleurs coups possible pour répondre au joueur humain. Cette intelligence a été développée dans la mesure de nos connaissances et du temps qui nous était donné.

Aide au joueur Notre programme est aussi doté d'une aide au joueur qui montre à l'utilisateur tous les mouvements qu'une pièce peut réaliser. Cette même aide indique aussi à un joueur quand il est en échec ou quand un joueur a gagné.

2.4 Interface Graphique

Pour pouvoir accéder à toutes ces fonctionnalités, nous avons réalisé une interface graphique.

2.4.1 Menu d'accueil

Sur le menu d'accueil, l'utilisateur peut sélectionner différentes options :
- Jouer contre un autre joueur ou joueur contre l'ordinateur.
- Choisir le plateau sur lequel jouer.

Le une fois les options choisies, le joueur peut alors lancer la partie.

2.4.2 Fenêtre de jeu

Quand la partie est lancée, la fenêtre de jeu apparaît. La fenêtre de jeu possède plusieurs fonctionnalités :

— Le plateau de jeu est affiché.
— Il est possible de déplacer les pièces à l'aide de la souris uniquement.
— Les mouvements qu'une pièce peut effectuer sont affichés quand une pièce est sélectionnée.
— Les pièces prises par un joueur sont affichées.
— Un annonceur indique quand un joueur est en échec / le joueur gagnant.

3 Spécification

Cette section contient les caractéristiques des types de sol que nous avons ajouté au jeu.

3.1 Plaine

La plaine est le type de sol de base du plateau. Sur ce type de sol, les déplacements de toutes les pièces sont ceux du jeu XiangQi traditionnel.

FIGURE 3 – Texture de la plaine

3.2 Palais

Le palais est le seul type de sol sur lequel les Généraux et les Conseillers peuvent se déplacer. Les autres pièces peuvent y entre et en sortir librement.

FIGURE 4 – Texture du palais

3.3 Sable

Le sable est utilisé dans le plateau "Désert". Sur le sable, les déplacements des Chariots sont limités à une case verticalement et horizontalement.

FIGURE 5 – Texture du sable

3.4 Montagne

La montage est utilisé dans le plateau "Montagne". Si un Canon est placé sur une case montagne, il peut se déplacer et aller manger une pièce adverse sans se soucier du nombre de pièces qu'il doit d'abord sauter.

FIGURE 6 – Texture de la montagne

3.5 Glace

La glace est utilisé dans le plateau "Glacier". Ce type de sol n'apport pour l'instant aucun mouvement spécifique aux pièces. Ce type de sol n'a pas encore été bien implémenté.

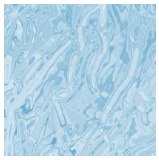


FIGURE 7 – Texture de la glace

4 Conception et Réalisation

Dans cette partie sont expliqués les différents mécanismes et représentations théoriques que nous avons développés pour réaliser le moteur du jeu. Les différents aspects techniques : classes et algorithmes sont abordés d'un point de vue général et théorique. Pour plus de précisions sur le fonctionnement des algorithmes, voir la Javadoc.

4.1 Conception IHM

Commençons par décrire les différents interfaces avec lesquelles l'utilisateur interagit (on rappel que ces différents éléments et leur fonctionnement sont décrits dans le manuel en 5.1 et 5.2).

4.1.1 Ecran d'accueil

Les fonctionnalités de l'écran d'accueil sont décrites dans le manuel d'utilisation (voir 5.1). Nous nous intéressons ici à la conception technique. La conception de cet écran (de classe GraphicMainMenu) est assez simple comme vous pouvez le voir sur la Figure 9.

FIGURE 8 – L'écran d'accueil comme vu par l'utilisateur.

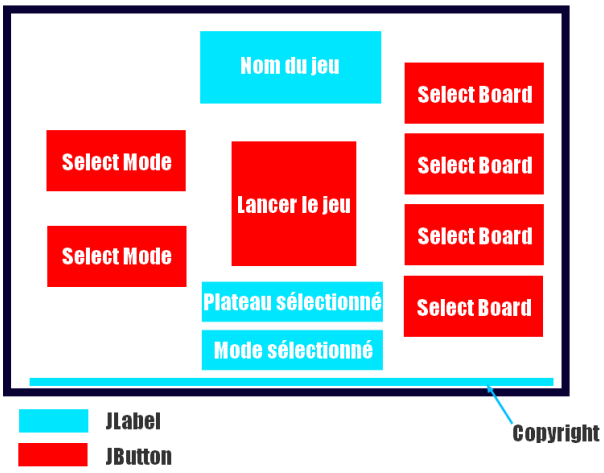


FIGURE 9 – Elements de l'IHM de l'écran d'accueil.

L'écran d'accueil est composé principalement de JButtons, les actions qui leurs sont attribuées sont décrites en Figure 9. Les boutons de gauche servent à sélectionner le mode de jeu (IA ou PvP) tandis que les boutons de droite ont pour effet de changer le plateau sélectionné. Appuyer sur un des boutons d'option provoque l'affichage de l'option sélectionnée dans les deux JLabel en bas au centre de la fenêtre. Le bouton "jouer" au milieu ("Lancer Jeu" en Figure 9), provoque l'envoi des options sélectionnées au moteur du jeu et crée/affiche la fenêtre de jeu.

4.1.2 Ecran de jeu

L'écran de jeu s'affiche quand le bouton "jouer" a été pressé dans l'écran d'accueil, c'est sur cette fenêtre que le ou les joueurs peuvent jouer. Les différentes fonctionnalités de cette fenêtre sont décrites dans le manuel d'utilisation (voir 5.2). La fenêtre de jeu (de classe BoardGUI) est d'une conception plus élaborée que l'écran d'accueil. En effet, on peut voir sur la Figure 11 que la fenêtre est principalement constituée d'éléments graphiques que nous avons créés : les objets de classe PlayerBoard et BoardGraphic.

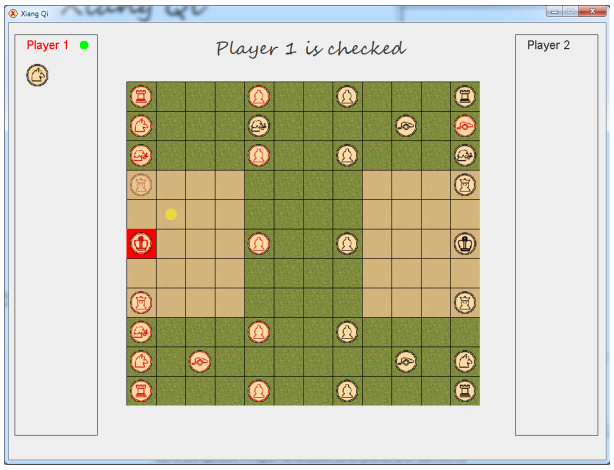


FIGURE 10 – L'écran de jeu comme vu par l'utilisateur.

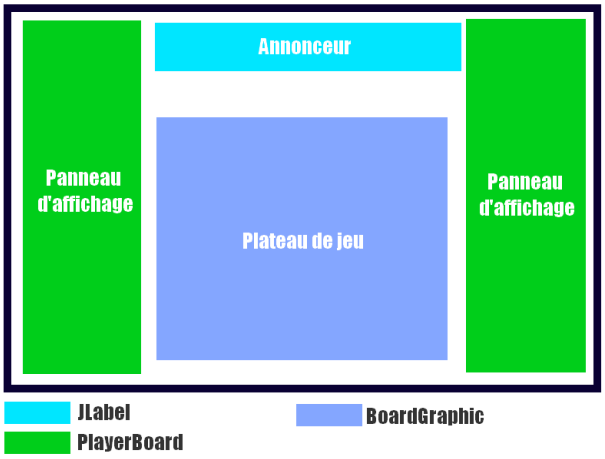


FIGURE 11 – Elements de l'IHM de l'écran de jeu.

BoardGUI C'est ce que nous appelons la "fenêtre de jeu". La classe BoardGUI est une classe fille de la classe JFrame, elle contient tous les éléments graphiques suivants :

- 2 PlayerBoard
- 1 BoardGraphic
- 1 JLabel

Tous ces éléments sont placés dans le contentPane de BoardGUI.

PlayerBoard La classe PlayerBoard est une classe fille du JPanel, sa particularité est qu'elle possède un attribut de type Player qui permet de garder en mémoire le joueur auquel un panneau créé a été associé. Une fois créé, il suffit juste d'appeler la fonction repaint() sur le panneau pour actualiser l'affichage. Tous les éléments affichés dans les panneaux PlayerBoard sont dessinés à partir des données du joueur en utilisant l'outil Graphics de Java.

BoardGraphic La classe BoardGraphic est ce que nous appelons le plateau de jeu. C'est une classe fille de la classe JPanel. Cette classe utilise le pattern Singleton, ceux parce que nous n'avons besoin que d'un seul plateau de jeu. Cette solution nous facilite aussi la tache quand il s'agit d'accéder à l'instance de BoardGraphic et à ses différentes méthodes (notemment repaint()). Ce n'est peut-être pas la meilleure utilisation de ce pattern mais elle a simplifié le développement.

La particularité de cette classe BoardGraphic est qu'elle est muni d'un certain nombre d'actions de type MouseListener. En effet, pour pouvoir déplacer les pièces (voir manuel d'utilisation 5.3.2), nous avons créé un système de zones de clic. Le fonctionnement de ce système repose sur 3 types de classe (Figure 12) :

- ActionZone : implémente l'interface MouseListener, cette classe possède des attributs qui contiennent les données concernant la zone d'effet de l'action (x1,y1,x2,y2). Elle mutualise le code de ces attributs pour les deux autres classes d'actions.

— SelectPieceActionZone : hérite de ActionZone, elle a pour fonction de détecter un clic dans une zone donnée de BoardGraphic. Cette action est responsable de l'action "prendre une pièce" (dans le but de la déplacer). Quand cette action détecte un clic, elle génère des zones d'action de type DeplacementActionZone en fonction des déplacements possibles de la pièces sélectionnée.
— DeplacementActionZone : hérite de ActionZone, elle a pour fonction de détecter le relâchement d'un clic dans une zone donnée. Cette action est responsable de l'action "poser une pièce" (après l'avoir prise). Cette action réalise aussi la connexion entre l'IHM et le moteur du jeu en informant le moteur du jeu du mouvement qui a été effectué.

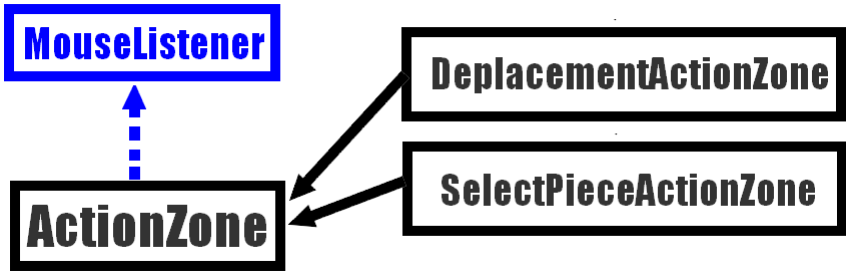


FIGURE 12 – Diagramme UML des classes d'action

Au panel BoardGraphic ne sont associées que des actions SelectPieceActionZone et DeplacementActionZone. Chacunes de ces actions est placée sur une case du plateau. Par exemple, quand le joueur de couleur noir a finit de jouer, le tour va passer au joueur de couleur rouge. A ce moment, le moteur va appeler repaint() sur l'instance de BoardGraphic. Cela aura pour effet de générer automatiquement des zones d'action de type SelectPieceActionZone sur toutes les cases contenant une pièce de la couleur rouge. Maintenant, joueur rouge va vouloir jouer. Ce dernier va donc aller cliquer sur une pièce de sa couleur. L'action SelectPieceActionZone qui avait été mise sur la case au repaint() va alors détecter le clic et générer des zones d'action de type DeplacementActionZone au dessus de toutes les cases où la pièce peut se déplacer (ces cases sont aussi marquées d'un indicateur (voir 5.4.1). Le joueur peut alors déplacer son curseur au dessus de la case où il souhaite déplacer la pièce (tout en maintenant le clic de la souris). Enfin, le joueur relâche son clic sur la case, cette action est alors détectée par l'action DeplacementActionZone qui va comptabiliser le mouvement et en informer le moteur du jeu. Toutes les actions sont alors retirées du panel de BoardGraphic et on commence un nouveau tour.

Remarque : Bien entendu, au cours de ce procédé, tous les indicateurs nécessaires sont affichés dynamiquement à l'écran par les différentes actions ainsi que par le moteur.

Annonceur L'annonceur, situé en haut de la fenêtre est simplement un JLabel, mis à jour au début de chaque nouveau tour en fonction des données courantes du match (échecs et joueur gagnant).

4.1.3 Les interractions IHM - Moteur

Voici un bref récapitulatif des interractions entre l'IHM et le moteur du jeu (expliquées en 4.1).

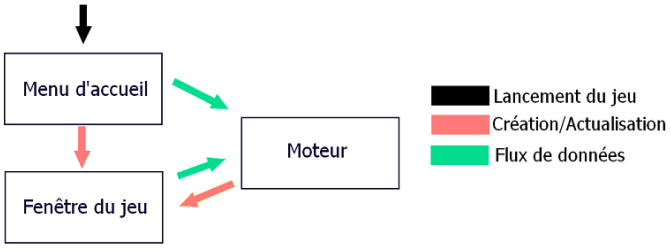


FIGURE 13 – Interractions entre les composantes de l'IHM et le moteur

Comme nous pouvons le voir en Figure 13, au lancement du jeu, l'écran d'accueil apparaît. Sur cet écran, il est possible de choisir des options. An lancement de la partie, les données représentant les options choisies sont transmises au moteur ; par la même action la fenêtre de jeu est créée. A ce moment, les joueurs vont pouvoir effectuer des actions dans la fenêtre de jeu (jouer), ces actions sont alors transmises au moteur. Ce dernier va les analyser, les traiter et enfin mettre à jour la fenêtre de jeu et la partie continue.

4.2 Classes de données de jeu

Les données de jeu sont les données relatives au jeu en lui même, c'est à dire le plateau et les pièces. Ces données varient peu au cours de la partie (on ne peut pas agir sur les types de sol en cours de partie par exemple).

4.2.1 La classe Ground

Un objet de type Ground représente un type de sol, en effet comme il est indiqué dans la partie 3, nous avons créé différents types de sol. Ces types de sol constituent la base du plateau de jeu. La particularité de tous ces types de sol est qu'ils peuvent tous être représentés de la même manière sous forme de données (d'où l'unique classe Ground). En effet, tous les types de sol possèdent deux caractéristiques :

— Un type : (String)
— Une texture : (File)

Une fois créés au lancement de la partie, les objets de type Ground ne sont pas modifiés au cours du jeu.

4.2.2 La classe Piece

Dans cette partie, il est plus pertinent de parler d'objet Piece, donc de pièces, plutôt que de la classe Piece en elle-même. En effet, un objet de type Piece représente un pièce du plateau. De la même manière que les types de sol, tous les types de pièces peuvent être définis par une même représentation de données, d'où la classe Piece. En effet, toutes les pièces possèdent les mêmes caractéristiques de données ; toutes les pièces ont :

— Un type : (String)
— Une texture : (File)
— Une couleur : (String, RED ou BLACK)
— Des coordonnées sur le plateau : (Coordinates)
— Des paternes de mouvement : (MovementPatern)
— Des mouvements possibles : (ArrayList<Coordinates>)
— Une zone où la pièce peut manger des pièces énemies : (ArrayList<Coordinates>)
— Un ensemble de règles qui régissent les déplacements : (ArrayList<Rule>)
— Un poids qui atteste de l'importance de la pièce : (int)
— Un indicateur de condition de victoire (en l'occurence, dans le XiangQi, seul les Généraux portent la condition de victoire) : Boolean

Au cours d'une partie, seul les coordonnées sur le plateau, les mouvements possibles et la zone couverte par la pièce sont susceptibles d'être modifiés. Le reste des attributs n'est jamais modifié au cours du jeu. Les objets de classe Piece sont créés d'une manière particulière (voir 4.4.2).

4.2.3 La classe Board

Un objet de type Board est un plateau de jeu. Un plateau de jeu possède bien entendu des dimensions, 12x11 dans notre cas. De plus, plateau de jeu est constitué de deux couches :

— Un couche physique formée de types de sol : Ground[12][11]
— Une couche implicite qui contient les pièces posées sur le plateau : HashMap<Coordinates,Piece>

Dans notre programme, il est possible de créeer plusieurs objets de type Board car nous avons par exemple besoin de créer des répliques du plateau pour évaluer les déplacement (nous y reviendrons plus tard). Cependant, dans un objet de type Board, seul la couche implicite (les unités) peut être modifiée ; la couche physique (le sol) n'est jamais modifiée.

4.3 Classes de données de la partie

Les classes de données de la partie contiennent des informations générales sur la partie et son déroulement/ses conditions. Une fois initialisées, ces données sont peu modifiées dans l'ensemble.

4.3.1 La classe Player

La classe Player contient toutes les informations relatives à un joueur ; c'est à dire :
— Le nom du joueur : (String)
— La couleur du joueur : (String RED ou BLACK)
— Les pièces avec lesquelles il peut jouer : (ArrayList<Piece>)
— Les pièces qu'il a perdues : (ArrayList<Piece>)
— Les pièces qu'il a capturées : (ArrayList<Piece>)
— Un indicateur qui dit si le joueur est en échec : (Boolean)

Les données des joueurs sont exploitées notemment par l'IHM, par exemple pour afficher les pièces prises par un joueur (voir 4.1.2).

4.3.2 La classe Match

La classe Match contient toutes les informations à propos de la partie qui est en train d'être jouée. Cette classe est une instance (paterne Singleton), en effet, nous n'avons pas besoin d'instancier plusieurs matchs en même temps puisqu'une seule partie peut être jouée à la fois. De plus, ce patern nous permet d'accéder aux informations concernant la partie facilement grace aux types statiques. La classe Match contient :
— Le premier joueur : (Player)
— Le deuxième joueur, susceptible d'être une IA : (Player/Chesster)
— Le plateau de jeu : (Board)
— Le joueur qui est en train de jouer : (Player)
— Le joueur qui attend son tour : (Player)
— Le joueur gagnant : (Player)
— Le nombre de tours : (int)

La classe Match a été conçue pour faciliter l'accès aux données de la partie par le moteur. Cette classe contient la méthode swap(), utilisée pour échanger le joueur qui joue et le joueur qui attend pour ainsi générer les tours.

4.4 Compilation des données de démarage

Au lancement de la partie, le jeu doit créer un certain nombre d'objets (Piece, Ground, Board) pour initialiser la partie. Or ces objets comme nous l'avons dit précédemment sont des représentations générique, elles ne sont que des squelettes. Il faut donc tout d'abord charger un certain nombre de données dans le programme afin de pouvoir créer ces objets. Ces données sont stockées dans différents fichiers texte :
— groundsettings.txt : contient toutes les données concernant les types de sol.
— piecesettings.txt : contient toutes les données concernant les types de pièces.
— boardsettings.txt : contient toutes les données servant à l'initialisation du plateau de jeu.

Remarque : Il existe en fait 4 fichiers boardsettings.txt, un pour chaque plateau que le joueur peut choisir dans le menu de démarage (voir 5.1.2).

Ces fichiers contiennent donc les informations utiles à l'initialisation du jeu / à la création de différents objets. Cette solution technique as pour avantage que toutes les données concernant chaque type d'objet Ground ou Piece sont regroupées dans un même fichier. Ce fichier est écrit avec une synthaxe particulière, les données sont donc lisibles et modifiables à volonté pour quiconque maître la grammaire de ces fichiers. Cependant, cette solution pose un inconvénient qui est qu'il faut compiler ces fichier, il faut extraire les données de ces fichiers textes. C'est pour cela que nous avons développé un groupe de 3 compilateurs afin de récupérer en 3 étapes toutes les données nécessaires à l'initialisation du jeu.

Remarque : Nous avons inventé le language dans lequel les fichiers de paramètres sont écrits. C'est un language qui s'écrit ligne par ligne. La grammaire de ce language est visible dans les classes de donnée du package compiler.lang (particulièrement les classes Grammar, RuleGrammar et BoardGrammar).

4.4.1 Etape 1 : GroundSettingCompiler

Ce premier compilateur va lire les informations contenues dans le fichier groundsettings.txt. Ce fichier contient toutes les informations pour créer des objets de type Ground c'est à dire :

- Les types de sol
- Leur texture graphique associée

Les données lues dans le fichier texte, si elles ne comportent pas d'erreur de grammaire, sont alors utilisées pour créer des objets Ground ou cours de la compilation du fichier. Ces objets sont stockés dans un répertoire appelé GroundsRepository. Ce dernier répertorie tous les types de sol déclarés dans le fichier texte. Par la suite, quand nous allons vouloir créer des objets Ground par exemple pour créer le plateau du jeu, nous utiliserons un constructeur dans lequel il nous suffit juste de spécifier le type de sol que nous voulons créer. Le constructeur ira alors chercher tout seul la texture associée au type de sol demandé dans le GroundsRepository ; par la même occasion, si le type demandé n'existe pas, le constructeur retourne une erreur. Ce mécanisme est partie intégrante du procédé de compilation.

4.4.2 Etape 2 : PieceSettingsCompiler

Ce second compilateur va, comme le précédent, lire les données contenues dans un fichier texte, ici le fichier piecesettings.txt. Avec ces données, le compilateur va créer des objets PieceModel. Les objets Piece sont des objets plus complexes que les objets de type Ground, pour éviter d'avoir à traiter trop de données d'un coup pendant la compilation, on utilise donc ces objets de type PieceModel. Un objet de type PieceModel regroupe toutes les informations concernant un type de piece tel qu'il est déclaré dans le fichier texte :

- Type de la pièce
- Textures pour chaque couleur (RED et BLACK)
- Tous les mouvements associés au type de piece, quelque soit la couleur de la pièce, le sol sur lequel elle repose etc...
- Les règles qui régissent les déplacements
- La condition de victoire

Cependant, il ne suffit pas de lire les informations dans le fichier texte pour pouvoir les exploiter directement. Nouys avons créé une synthaxe qui permet d'écrire facilement les données concernant chaque type de pièce. C'est à dire que, par exemple pour programmer les mouvements, il existe un certain nombre de fonctions de génération qui rendent la déclaration plus aisé. Le problème c'est qu'il faut maintenant comprendre ces différentes fonctions de génération afin de générer les mouvements. Le PieceSettingsCompiler délègue la génération des données de mouvement au deux module : PathBuilder et MovementPaternBuilder. De même, pour compiler les règles associées aux pièces, qui elles aussi sont déclarées en utilisant un certain nombre de fonctions de génération, le PieceSettingsCompiler délègue la tache au RuleBuilder. Nous n'entrerons pas dans les détails sur ces différents points.

Une fois créés, les objets PieceModel sont, de la même manière que les objets Ground stockés dans un répertoire appelé PiecesRepository. Ainsi, d'après le même mécanisme que pour la création simplifiée des objets Ground, il nous suffira par la suite d'utiliser un constructeur spécial pour créer des objets Piece. Nous n'aurons besoin que d'indiquer :

— Le type de la pièce
— La couleur de la pièce
— La position de la pièce

Le constructeur ira alors de lui-même chercher dans le PiecesRepository l'objet PieceModel de type correspondant à celui que l'on désire créer et en extraira les données pertinentes (texture de la bonne couleur, déplacements etc...).

Remarque : Cette étape intervient après la compilation des types de sol car certains déplacements peuvent être dépendants du type de sol sur lequel la pièce repose. Il est donc nécessaire de les connaître à l'avant pour pouvoir vérifier si ils existent bien et que le fichier n'est pas mal écrit.

4.4.3 Etape 3 : BoardSettingsCompiler

Cette dernière étape intervient après la compilation des types de sol et des types de pièces car elle vise à générer le plateau de jeu qui a besoin de ces différents éléments. Toujours de la même façon que les deux précédentes étapes de compilation, le BoardSettingsCompiler va lire dans le fichier texte contenant les données du plateau que l'utilisateur a choisit à l'écran d'accueil (voir 5.1.2). Ceux afin de charger dans le programme les données concernant le plateau de jeu. Grace aux deux précédentes étapes de compilation, le fichier contenant les informations concernant le plateau est lisible et compréhensible. De la même façon que pour écrire les paramètres des pièces, on peut utiliser quelques fonctions qui permettent de générer plus facilement le terrain par exemple (voir Javadoc, classe BoardGrammar).

Enfin, les informations chargées pendant la compilation du plateau sont placées dans un objet de type Board. Cet objet va ensuite être placé dans la classe Match en tant que plateau de jeu (voir 4.3.2). Si la compilation s'est passée sans erreurs, le jeu peut être lancé.

4.4.4 Bilan de la compilation

La compilation est une étape important de l'exécution du jeu, elle est effectuée au lancement de la partie. Elle permet de charger dans le programme toutes les données nécessaires au déroulement du jeu (objets Ground, Piece, Board). Elle facilite la compréhension des différents types d'objets (Ground et Piece) intervenant dans le jeu et permet la modification ou l'ajout de nouveaux objets dans le jeu. La compilation se passe en 3 temps, comme il est récapitulé en Figure14.

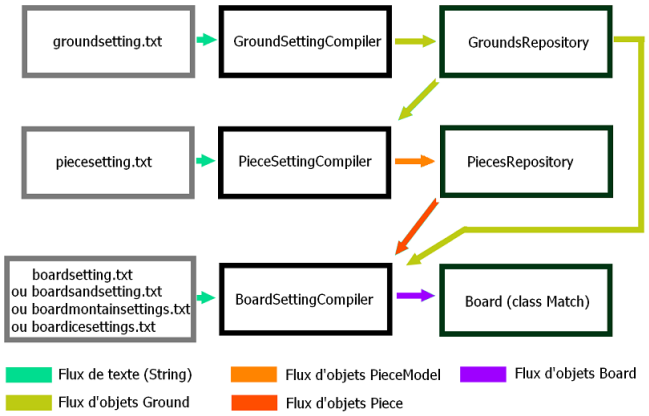


FIGURE 14 – Interractions des différents modules de compilation

Le choix d'un de créer cette étape assez complexe a été réalisé car il rend notre programme plus modulable et facilite le traitement et le développement. Cela ne nous était pas imposé mais nous avons pensé que cela pourait être un bel ajout à notre programme en plus d'être enrichissant.

4.5 Représentation des mouvements d'une pièce

Nous avons vu précédemment les différents attributs que contient la classe Piece et comment ils sont créés. Nous n'avons cependant pas abordé un point important : la représentation des mouvements des pièces. En effet, les mouvements des pièces sont représentés par un type de données particulier : le MouvementPatern. Un MouvementPatern est constitué des attributs suivants :

- Une liste de chemins que la pièce peut parcourir pour bouger : (ArrayList<LinkedList<Coordinates»). Nous reviendrons sur ce point un peu plus tard.
- Un ensemble de tags, regroupés dans une classe (MovementTag).

MovementTag Un MovementTag est un ensemble de tags qui donnent des informations sur la nature d'un déplacement et sur le contexte dans lequel il doit être employé. Plus précisément, il contient :

- Un attribut qui représente la couleur de la pièce par lequel il doit être employé. (String)
- Un attribut qui représente le type de pièce par lequel il doit être employé. (String)
- Un attribut qui exprime si les mouvements contenus dans le paterne sont standards ou alternatifs. (Nous y reviendrons).
- Un attribut qui exprime sur quel type de sol les mouvements contenus dans le paterne doivent être employés.

La plupart des informations contenues dans le MovementTag ne sont utiles qu'à la création des objets Piece. Les attributs de couleur et de type de pièce servent en fait qu'à dire au constructeur de Piece que ces informations concernent l'objet Piece que l'on désire créer (et donc extraire les bonnes informations des objets PieceModel). Pour plus de précisions, voir la Javadoc (classes Piece, TaggedPath, MovementPaternBuilder). Les données qui nous importent quand nous parlons de représentation des mouvements sont la spécification (mouvement standard ou alternatif) et le type de sol sur lequel le mouvement est valide.

Déplacements en représentation chaînée Nous l'avons vu, les MovementPatern possèdent en fait une liste de chemins que la pièce à laquelle le paterne est attribué peut emprunter dans un certain contexte (donné par le MovementTag). Les chemins sont représentés sous forme de listes chaînées d'objets Coordinates (des coordonnées x,y). Ces coordonnées sont en fait traitées comme des incréments. Vous pouvez voir un exemple en Figure 15.

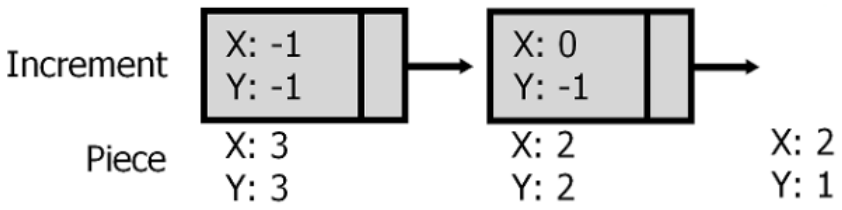


FIGURE 15 – Calcul d'un chemin

On récupère dans un premier temps les coordonnées courantes de la pièce, puis on ajoute la valeur de chaque composante x et y des coordonées contenues dans la liste aux composantes x et y des coordonnées de la pièce. De ce fait, on arrive en ajoutant successivement aux coordonnées courantes de la pièces, les valeurs x et y des coordonnées contenues dans la liste à générer un déplacement positions par positions, relatif à la position initiale de la pièce.

Nous avons eu besoin de ce mécanisme, car rappelons-le, dans le XiangQi certaines pièces ne peuvent pas sauter par dessus d'autres pièces. Nous devons donc être capable de regarder pour chaque étape du déplacement si la case est libre ou non afin de pouvoir générer les déplacements possibles de la pièce. Il n'était pas possible de se contenter d'un algorithme qui regarde simplement par un calcul mathématique si le chemin est libre ou non car certaines pièces (en particulier les Chevaux) ont des déplacements plutôt exentriques qui ne peuvent pas être évalués par les maths. En plus, un avantage de ce système est qu'il nous permet de représenter nimporte quel mouvement sur le plateau. Ce qui est utile car, comme nous l'avons vu précédamment, il nous est possible de créer nimporte quelle pièce dans le fichier texte piecesettings.txt. Avec cette représentation des déplacements nous avons donc la possibilité de réaliser un moteur qui peut gérer nimporte quelle sorte de déplacements aussi exentriques qu'ils soient.

4.6 Règles de déplacements

Les règles de déplacement sont représentées par des classes qui héritent toutes d'une superclasse abstraite Rule. Elles contiennent les données nécessaires à la validation de règles qui sont propres à la pièce à laquelle elles sont associées. La superclasse Rule contient une donnée essentielle au traitement des règles par le RuleEvaluator (voir 4.7.3), la priorité d'évaluation. Cette donnée est un int qui sert à déterminer à quelle étape d'un déplacement cette règle doit être évaluée. Il y a 3 valeurs de priorité différentes :

- Priorité 3 : ces règles sont à évaluer en premier, elles peuvent directement invalider le déplacement en fonction des conditions initiales.
- Priorité 2 : ces règles sont à évaluer en second, elles régissent la validité du chemin d'une pièce.
- Priorité 1 : ces règles sont à évaluer en dernier, elles disent si l'emplacement final du déplacement est valide.

Dans le jeu XiangQi, on compte 3 règles importantes :

JumpRule De priorité 2, la JumpRule, littéralement, la règle de saut dicte si une pièce peut sauter par dessus d'autres pièces et quand elle peut manger une pièce adverse. Cette règle est spécifiquement utilisée pour les Cannons qui doivent obligatoirement sauter par dessus une autre unité avant de pouvoir manger une pièce énemie.

FinalPositionGroundRule De priorité 3, cette règle est utilisée pour restreindre les déplacements de certaines pièces à un type de sol spécifique. Par exemple, les Conseillers ou le Général ne peuvent pas se trouver sur un type de sol différent du palais sur lequel ils sont initialement placés. Cette règle assure donc qu'après le déplacement, la pièce se trouve sur un type de sol autorisé.

PieceTypeAlignmentRule De priorité 3, cette règle assure qu'à la fin d'un déplacement, une pièce n'est pas alignée avec un certain type de pièce. Cette règle est utilisée spécifiquement pour les Généraux. En effet, on le rappel, les deux Généraux rouge et noir ne peuvent pas se trouver face à face sur la même ligne.

Les pièces contiennent toutes une liste de règles. Une pièce peut avoir des règles spécifiques si elles sont déplarées dans le fichier texte piecesettings.txt, sinon la pièce est soumise aux règles standards (classe BaseRules). Les règles servent donc à limiter les déplacements des pièces sur le plateau mais aussi à donner des spécificités / avantages à certaines pièces. Les règles sont évaluées par le RuleEvaluator (voir 4.7.3).

4.7 Moteur du jeu

La première fonction du moteur du jeu était l'étape de compilation des données (voir4.4). Cette étape, bien qu'importante ne participe qu'à l'initialisation de la partie. Le parties du moteur du jeu que nous allons décrire maintenant sont celles responsables du déroulement du jeu. Notre solution

technique pour réaliser cette partie du moteur a été de se dire que tous les calculs de déplacements possibles peuvent être réalisés à l'avance entre chaque tour. En effet, quand un joueur déplace un pièce sur le plateau, le coup qu'il joue influe non seulement sur la pièce déplacée mais aussi sur les autres pièces. Il faut donc après chaque tour recalculer tous les mouvements possibles. Mais avant cela, parlons de la partie du moteur la plus "haute", celle qui gère les tours et les évènements (échecs / joueur gagnant).

4.7.1 MainEngine

La classe de traitement MainEngine comporte une méthode principale appelée newTurn(). Comme son nom l'indique, cette méthode est appelée pour générer un nouveau tour / passer la main au joueur adverse. La fonction de cette partie du moteur est dans un premier temps de mettre à jour les mouvements que le joueur énemis peut effectuer. Pour se faire, MainEngine fait appel à une partie plus "basse" du moteur le MovementValidator, nous reviendrons sur ce point par la suite. En gros, cette partie du moteur est chargée de calculer tous les mouvements qu'une pièce donnée peut effectuer. MainEngine va donc d'abord réévaluer tous les déplacements de toutes de les pièces du joueur qui vient de jouer. De ce fait, MainEngine va pouvoir détecter si, par son action, le joueur qui vient de jouer a mis en échec le joueur adverse. Par conséquent d'une part, MainEngine va mettre à jour les indicateurs de l'IHM (voir 5.4) et d'autre part, évaluer les mouvements du joueur qui s'apprette à jouer en prenant en compte les mouvements possibles de son adversaire.

Il est nécessaire d'évaluer les mouvements dans cet ordre afin de pouvoir bien gérer les échecs par exemple. Si un joueur est en échec, il faut obligatoirement que sont prochain coup le sorte de sa situation d'échec (sinon le joueur que a mis en échec a gagnée). Contrairement, si un joueur n'est pas en échec, il ne doit pas pouvoir et il ne veut se mettre en échec lui-même, il faut donc "épurer" les mouvements que ses pièces peuvent réaliser pour ne pas qu'il se mette en échec lui-même. C'est pourquoi il est nécessaire d'évaluer les mouvements du joueur qui vient de joueur avant ceux du joueur qui s'apprête à jouer. Pour pouvoir "épurer" les mouvements indésirables, nous utilisons un objet de classe Simulation.

Simulation Un objet de classe Simulation est constitué d'un plateau (classe Board) ; ce plateau est initialisé grace à la méthode reset(). Le principe de la Simulation est de réaliser une copie du plateau de jeu et d'y faire un mouvement grâce à la métode simulate(). Cette même méthode va, par le biais du MouvementValidator réévaluer tous les mouvements de toutes les pièces présentes sur le plateau avec le mouvement qui a été réalisé. Si ce mouvement en question met le joueur qui l'a effectué en échec ou non, la méthode simulate() retourne un booléen. Ce booléen peut ensuite être exploité par la méthode qui a appelé la simulation.

Une fois les mouvements de toutes les pièces recalculés, si aucun joueur n'est déclaré gagnant, MainEngine appel la fonction swap() de la classe Match pour inverser le joueur qui joue et le joueur qui attend, ainsi générant un nouveau tour. Enfin, MainEngine va actualiser tous les éléments de BoardGUI (la fenêtre de jeu) pour montrer les pièces à leur emplacements correct, les pièces prises, les zones d'action etc...

4.7.2 MovementValidator

Nous avons vu dans la partie précédente que MainEngine appelle la fonction MovementValidator pour évaluer les déplacements d'une pièce donnée. Le MovementValidator n'évalue en pas réellement les mouvements lui-même, il fait dans un premier temps appel au RuleEvaluator qui est la couche la plus basse du moteur, nous y reviendrons plus tard.

Le rôle du MovementValidator est dans un premier temps de récupérer la liste de MouvementPatern contenue dans l'objet Piece duquel on veut mettre à jour les déplacements. Parmis cette liste de MovementPatern, le MovementValidator va extraire celui/ceux qui doivent être utilisés sur le type de

sol sur lequel la pièce repose. On le rappelle, les pièces peuvent avoir des mouvements différents selon le type se sol sur lequel elles sont posées. Cette sélection est réalisée grace aux MovementTags. Le MovementValidator va alors évaluer les déplacements contenus dans ces différents MovementPatern. Pour se faire, il fait d'abord appel au RuleEvaluator. Ce qu'il faut pour l'instant retenir de cette dernière est qu'elle évalue la validité d'un déplacement par rapport aux règles qui sont associées à la pièce et retourne un Integer en fonction de la validité du mouvement. Le MovementValidator va alors se charger d'interprêter cette valeur de retour et placer la destination finale du déplacement (voir 4.5) soit dans la liste possiblemoves soit dans la liste coveredzone de la pièce. Le déplacement peut aussi être concidéré comme non valide, dans ce cas les coordonnées finales du déplacement ne sont enregistrées nulle-part.

En réalité, le MovementValidator ne fait pas qu'interprêter les valeurs de retour du RuleEvaluator, il peut aussi influer sur la validité d'un déplacement. Par exemple, si la pièce dont on valide les déplacements et une pièce qui tient une condition de victoire (un Général par exemple), il est possible qu'un de ces mouvements soit valide vis à vis des règles de déplacement de la pièce mais non valide vis à vis de l'état actuel du jeu. Par exemple, un Général pourait très bien pouvoir se déplacer à une position, ce déplacement serait valide vis à vis des règles de déplacement du Général mais si une pièce ennemie couvre cette case, le joueur pourait se mettre en échec en jouant ce coup. Le MovementValidator va donc concidérer ce mouvement non valide et ne pas l'enregistrer.

4.7.3 RuleEvaluator

Le RuleEvaluator est la couche du moteur la plus éloignée des entrées de données de l'IHM. Cette partie du programme travaille essentiellement avec les informations de base contenues dans les objets Piece à savoir les paths (voir 4.5) et les règles de déplacement associées à cette même pièce.

Comme il l'a déjà été dit plus haut, le RuleEvaluator évalue la validité d'un déplacement donné en fonction des règles qui sont associées à cette pièce. Le MovementValidator passe donc en paramètre au RuleValidator, une file de priorité formée à partir des règles associées à la pièce ainsi qu'un déplacement à évaluer (Figure 16).

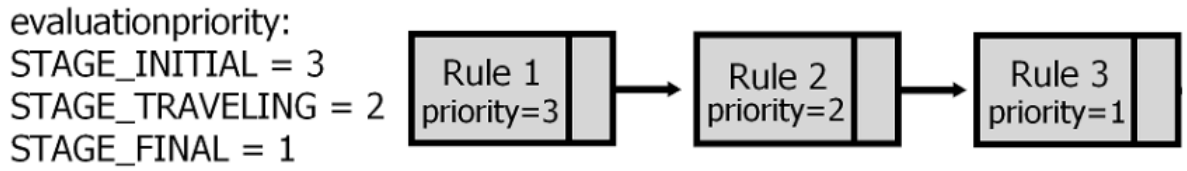


Figure 16 – Illustration d'une file de priorité de règles

La file de priorité est de type LinkedList<Rule>, or elle contient des types de règles différents (voir 4.6) . Nous avons donc recours à un paterne visiteur pour pouvoir évaluer efficacement chaque règle. En effet, le RuleEvaluator est un visiteur, classe qui implémente l'interface Evaluator et qui va visiter chaque règles pour pouvoir les traiter chacun différemment. Pour plus d'informations, voir Javadoc.

Quand le RuleEvaluator a évalué un déplacement à l'aide de la file de priorité, il retourne un Integer en fonction de la validité du mouvement. Les valeurs possibles sont :

- 0 - Le déplacement n'est pas valide.
- 1 - La pièce peut manger une pièce adverse en effectuant ce déplacement.
- 2 - La pièce couvre la zone mais ne peut pas s'y déplacer.
- 3 - Le déplacement conduit à la mise en échec du joueur adverse.
- 4 - La pièce peut se déplacer mais ne peut pas capturer une pièce adverse.

Pour plus de précisions sur les valeurs de validité voir la Javadoc (classe ValidityGrammar). Les valeurs peuvent alors être interprêtées par la méthode appelante, principalement par la classe MovementValidator.

4.7.4 Récapitulation du fonctionnement du moteur

La Figure 17 récapitule les informations entre les différentes parties du moteur du jeu :

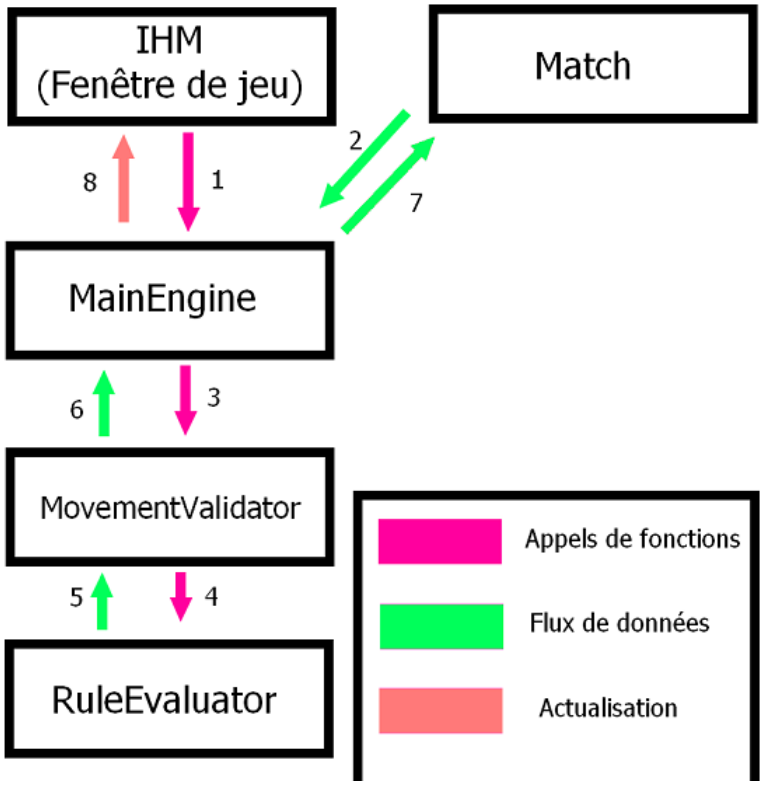


FIGURE 17 – Interractions au sein du moteur

1. Le joueur effectue une action, l'IHM informe MainEngine du mouvement qui a été effectué.
2. MainEngine récupère les données de Match (plateau, joueurs) afin de pouvoir revalider tous les mouvements pour préparer le tour du joueur suivant.
3. MainEngine valide les mouvements de toutes les pièces du plateau (récupérées à l'étape précédente) par le biais de MouvementValidator.
4. MouvementValidator va "sous-traiter" la validation des mouvements à RuleEvaluator.
5. RuleEvaluator retourne une valeur à propos de la validité des mouvement que MovementValidator lui a passé.
6. MovementValidator va envoyer à MainEngine les informations concernant notemment les échecs.
7. MainEngine va actualiser les données du Match à partir du retour de MovementValidator.
8. MainEngine va actualiser l'IHM et le nouveau tour peut commencer.

4.8 Chesster, l'intelligence artificielle

On rappel qu'il est possible de choisir de jouer joueur contre IA (voir manuel 5.1.1), nous allons vous la présenter. Baptisée Chesster, l'intelligence artificielle de notre jeu est capable de répondre aux mouvements d'un joueur humain adverse. D'un point de vue données, Chesster est une classe fille de la classe Player et peut être instanciée.

Idée L'idée derrière la conception de Chesster est que chaque pièce possède une importance dans le jeu. En effet, comme vous avez pu le constater dans la description de la classe Piece (voir 4.2.2), chaque pièce possède un poids représenté par une variable int. Chesster est capable d'évaluer chaque coup qu'il lui est possible de faire et de lui associer une importance en fonction des conséquences qu'entraîne le coup (échec, prise de pièce énemies). La génération du comportement s'effectue en plusieurs étapes :

1. Génération d'une liste de coups possibles : tous les coups possibles de toutes les pièces appartenant à Chesster encore en jeu sont associés à un poids (classe PriorityMove). Tous ces objets PriorityMove sont alors rassemblés dans une même liste.
2. Evaluation de la priorité de chaque mouvement : Tous les mouvements sont passés un à un dans une fonction de transition à la manière d'un perceptron (Figure 18). Un perceptron est un élément de traitement de données utilisé dans les réseaux de neurones. Originellement, il prend en entrée un certain nombre de valeurs (ici une, un mouvement) et passe les données dans une fonction de transition. Une fonction de transition est originellement un calcul mathématique qui va multiplier un nombre passé en entré par un poids (un autre nombre) déjà défini. Une fois que toutes les entrées ont été multipliées par leur poid qui leur est associé, on somme le résultat de tous les calculs et on se sert de cette somme pour générer un comportement. Avec Chesster, nous n'avons pas de nombre en entrée du "perceptron" mais un coup possible. Nous avons donc réalisé un algorithme qui va pouvoir associer un poids à chaque mouvement en fonction de ses conséquences. Par exemple, le meilleur coup possible serait un coup qui permet de prendre une pièce avec un poids fort (un pièce importante) tout en mettant le joueur adverse en échec et en étant sûr que la pièce jouée ne peut pas être prise au tour suivant. A l'inverse, un mauvais mouvement serait un mouvement qui ne permettrait pas de capturer une pièce adverse, ni de mettre le joueur adverse en échec. La fonction de transition de Chesster permet donc d'associer un poid à chaque mouvement qu'il lui est possible de faire à la manière d'un perceptron. (Voir la Javadoc de la classe Chesster pour plus de précisions).
3. Tri des mouvements possibles : une fois le poids de chaque mouvement évalué, il faut trier tous les mouvements pour en extraire le plus intéressant. Nous créons une file de priorité en fonction du poids associé à chaque mouvement.
4. Jouer un coup : grace à la file de priorité créée à l'étape précédente, il ne nous reste plus qu'à jouer le mouvement le plus intéressant, le premier coup dans la file.

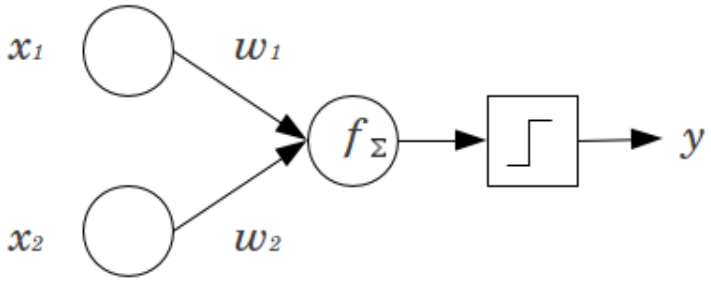


FIGURE 18 – Schéma d'un perceptron simple

Chesster est donc capable d'évaluer les mouvements les plus intéressants à l'aide de différents systèmes de poids. Cet algorithme fonctionne bien et est capable de répondre aux agressions d'un joueur humain en prenant en compte un coup à l'avance les conséquences de ses mouvements. Il n'est cependant pas capable d'élaborer des stratégies sur plusieurs coups à l'avance comme pourrait le faire une IA plus élaborée. Chesster a été développé dans les limites de nos connaissances et du temps qui nous a été alloué. Ce programme serait bien-sûr sujet à bon nombre d'améliorations.

5 Manuel Utilisateur

wrapfig

Cette section décrit l'utilisation du jeu par le ou les joueurs. Toutes les options et indicateurs sont illustrés et expliqués ici.

5.1 Ecran d'accueil

Le jeu, une fois lancé dans Eclypse affiche l'écran d'accueil (Figure 19).

FIGURE 19 – Menu d'accueil

5.1.1 Sélection du mode de jeu

Sur la gauche de la fenêtre, on peut voir les deux boutons servant à sélectionner le mode de jeu :
— "IA" set à sélectionner le mode de jeu où un joueur joue contre l'ordinateur (contre Chesster)
— "PvP" set à sélectionner le mode joueur contre joueur.

Par défaut, le mode de jeu PvP est sélectionné.

5.1.2 Sélection du terrain

Sur la droite de la fenêtre, on peut voir 4 boutons servant à sélectionner le plateau de jeu. Chaque plateau possède des particularités environnementales différentes. Par défaut, le terrain "Plaine" est sélectionnée.

5.1.3 Indicateurs d'option

En bas, au centre de la fenêtre sont affichées les options sélectionnées : en haut le terrain, en bas le mode de jeu.

5.1.4 Bouton "Jouer"

Au centre de la fenêtre se trouve le bouton "jouer" ayant pour apparence le logo du jeu, il sert à lancer la partie une fois les options choisies. Quand la partie est lancée, la fenêtre de jeu apparaît.

5.2 Ecran du jeu

L'écran du jeu (Figure 20) apparaît quand l'utilisateur appuie sur le bouton "jouer" sur l'écran d'accueil. Fermer la fenêtre de jeu fermera aussi l'écran d'accueil. L'écran du jeu est constitué des éléments décrits ci-dessous.

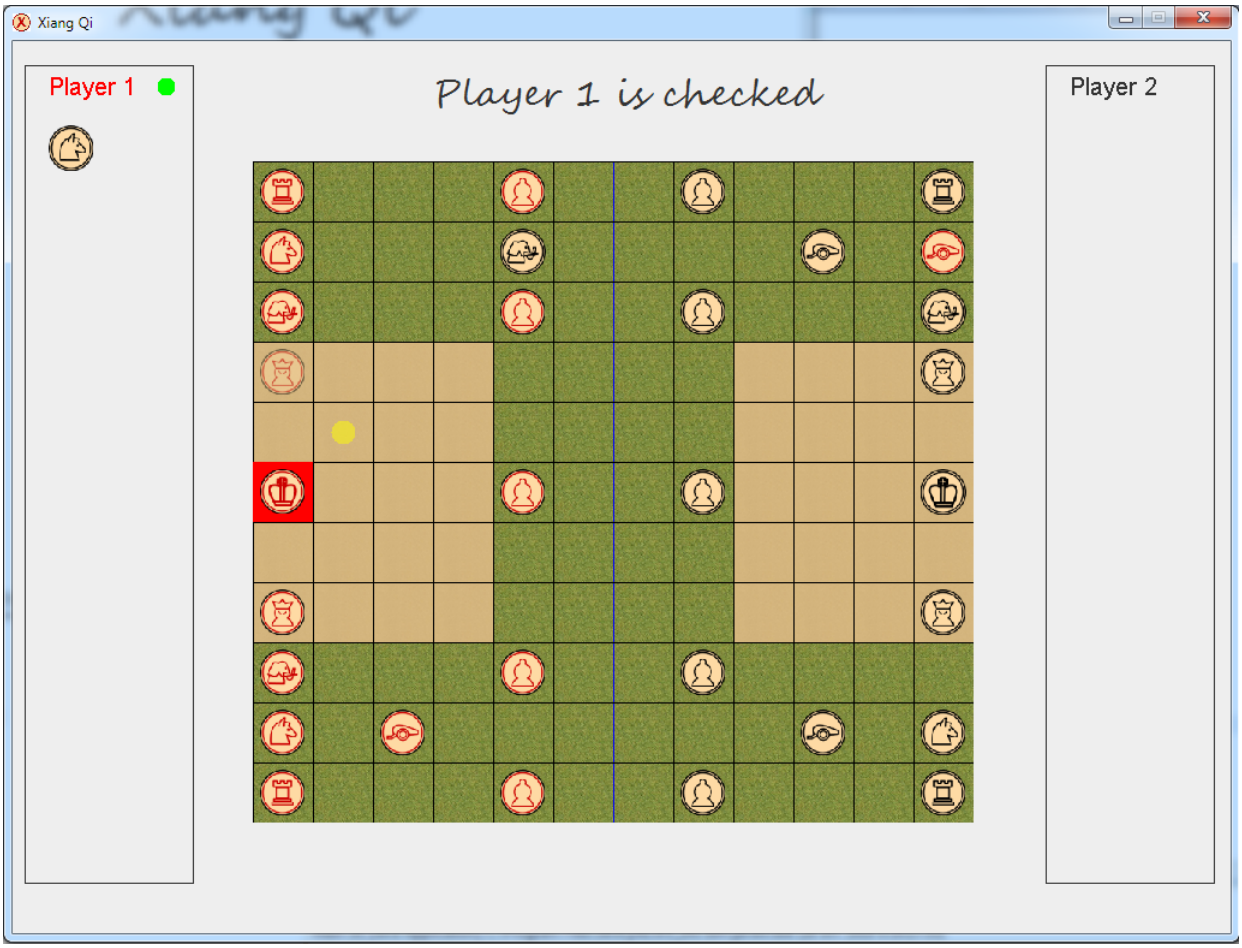


FIGURE 20 – Menu d'accueil

5.2.1 Le plateau de jeu

Le plateau de jeu est affiché au centre de la fenêtre, on peut y voir toutes les unités et spécificités de l'environnement. C'est aussi sur cette partie de la fenêtre que les joueurs vont pouvoir jouer.

5.2.2 Les panneaux d'affichage

Sur les côtés sont visibles deux panneaux d'affichage, un pour chaque joueur. Le nom de chaque joueur est affiché en haut du panneau. Dans ces panneaux seront affichés les pièces ennemies prises par chaque joueur.

5.2.3 L'annonceur

En haut de la fenêtre se trouve l'annonceur, sa fonction est d'indiquer les évènements importants au cours de la partie. Il peut indiquer :

— Quand un joueur est en échec.
— Quand un joueur a gagné.

Il disparaît quand il n'arrive aucun évènement important.

5.3 Comment Jouer

Après avoir sélectionné les paramètres de jeu voulus dans l'écran d'accueil, la fenêtre de jeu apparaît et il est alors possible de jouer.

5.3.1 Déroulement de la partie

Le XiangQi est un jeu qui fait intervenir deux joueurs de couleur différente : rouge et noir. Ces deux joueurs peuvent chacun déplacer une seule pièce par tour (si le mode de jeu qui a été sélectionné est "IA", l'ordinateur joue seul). Le but du jeu est, rappelons-le, le même qu'aux échecs occidentales, c'est à dire : mettre en échec le Général (le roi) adverse. Par conséquent, un joueur gagne quand le Général adverse est en échec, n'a plus de mouvements possibles et ne peut être sauvé par le déplacement d'une autre pièce. Dans ce cas le joueur est "mat", le joueur adverse a gagné et la partie s'arrête.

5.3.2 Commandes

Quand c'est au tour d'un joueur, le joueur peut déplacer une pièce et une seule. Pour ce faire, il suffit de :

1. Placer la souris au dessus de la pièce que le joueur souhaite déplacer (une pièce de sa couleur).
2. Cliquer sur la pièce et ne pas relâcher.
3. Glisser la pièce jusqu'à l'emplacement voulu.
4. Relâcher le clique.

Remarques :

— Si le joueur essaye de placer la pièce à un endroit interdit, la pièce revient à son emplacement de départ.
— Si le joueur qui joue est en échec, certains mouvements qui sont d'ordinaire possibles peuvent ne pas l'être si ils ne contribuent pas à débloquer la situation.
— A l'inverse, si un joueur n'est pas en échec, certains mouvements d'ordinaire possibles peuvent ne pas l'être si ils mettent le joueur en échec quand ils sont joués.

5.4 Les indicateurs visuels

La fenêtre de jeu possède un certain nombre de signaux visuels pour indiquer l'état actuel de la partie, voici leur description et leur fonctionnement.

5.4.1 Les signaux de déplacement

Comme il est visible sur la Figure 21, losrqu'un joueur tient une pièce dans sa main (voir 5.3.2), la pièce qu'il tient devient affichée en transparence sur le plateau. De plus, les cases dans lesquelles la pièce peut se déplacer sont marquées d'un point jaune transparent.

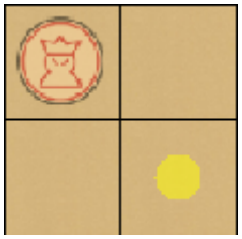


FIGURE 21 – Une pièce "en cours de déplacement"

Remarque Quand un joueur prend une de ses pièces pour la déplacer, le curseur de la souris prend l'apparence de la pièce sélectionnée. Cet effet n'est pas visible sur la Figure 21.

Lorsqu'une pièce peut manger une pièce adverse, l'indicateur n'est plus jaune mais bleu, (Figure 22).

FIGURE 22 – Illustration du marqueur "manger"

5.4.2 Les panneaux d'affichage

Les panneaux d'affichage sont situés sur les côtés de la fenêtre de jeu (voir 5.2.2). Une partie d'un panneau d'affichage est visible en Figure 23.

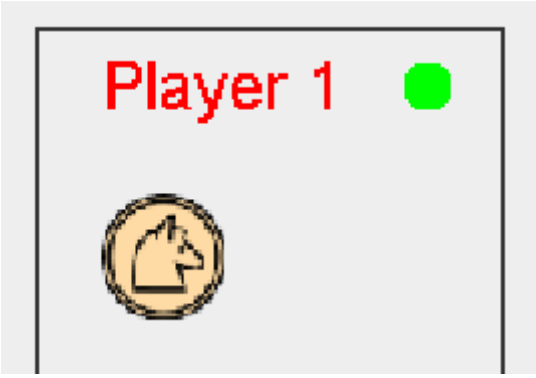


FIGURE 23 – Panneau d'affichage

Le panneau d'affichage donne 3 types informations :
- La couleur du joueur en indiquée par la couleur du nom du joueur (ici, nous pouvons voir le panneau du joueur rouge).
- Les pièces prises par le joueur sont affichées dans le panneau (ici, le joueur a capturé un Cheval).
- Le point vert à côté du nom du joueur indique que c'est à son tour de jouer.

5.4.3 Annonceur et signal d'échec

L'annonceur se trouve au dessus du plateau de jeu (voir 5.2.3). Il donne des informations sur l'état courant de la partie. Il a deux types de messages : les annonces d'échec (Figure 24) et les annonces de victoire (Figure 25).

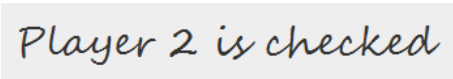


FIGURE 24 – Message d'échec

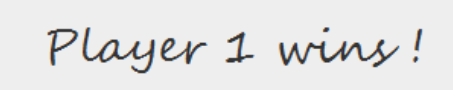


FIGURE 25 – Annonce de gagnant

En plus du message d'annonce, quand un joueur est en échec, la case sur laquelle le Général est posé s'affiche en rouge (Figure 26).

FIGURE 26 – Général en échec

6 Déroulement du projet

6.1 Répartition des tâches

Comme nous l'avions dit dans l'introduction, nous avons choisi ce projet car il présentait des aspects qui nous attiraient : Almamy CAMARA et Alexis CHOLLET voulaient du développement graphique tandis que Dorian CHENET recherchait plus de l'algorithmie. De plus, Dorian CHENET possédant, un niveau en programmation un peu plus élevé que ses deux collègues était plus à même de développer le moteur du jeu car plus à l'aise dans la théorie.

Dans un premier temps, tous les membres de l'équipe se sont attelés à analyser le sujet et à modéliser les premières classes de données. Ceci nous a pris environ 3 séances afin de pouvoir avoir un modèle de classes de données stable/à peu près définitif. Ensuite, nous nous sommes répartis les tâches de la manière suivante :

- CHENET Dorian commence à développer le moteur et les premiers algorithmes.
- CHOLLET Alexis et CAMARA Almamy se chargent de définir proprement les objectifs du projet ainsi que le mécanisme de gestion des tours.

Ensuite, une fois ces tâches de démarage ménées à bien, nous nous sommes répartis les tâche de la manière suivante :

- CHENET Dorian continue à développer le moteur du jeu.
- CHOLLET Alexis et CAMARA Almamy se chargent de développer un squelette d'IHM graphique.

Ces deux tâches, une fois menées à bien ont permis pour le premier point d'avancement d'avoir les parties algorithmiques et IHM bien avancées. Nous avons alors continué sur cette lancée avec la même répartition des tâches tout en approfondissant notre travail jusqu'au deuxième point d'avancement. L'enjeu de ce deuxième point d'avancement était de réussir à réunir la partie moteur et la partie IHM ; ce qui fut fait dans les temps.

Pour la suite du projet, après le deuxième point d'avancement, bien qu'il ne restait théoriquement pas grand chose à faire, nous avons plutôt mal répartit les tâches. L'IHM étant presque finie, le binôme CAMARA/CHOLLET était chargé de terminer les derniers détails, cependant ces détails bien que mineurs ont pris du retard et Dorian CHENET a donc dû s'en occuper en plus du moteur à développer. Ajoutez à cela que le moteur a été plus difficile que prévu à développer, la charge de travail a plus porté du côté algorithmique, donc sur la partie de Dorian CHENET. Ce dernier, ayant développé le moteur depuis le début a donc dû assurer seul le développement de la fin de l'IHM et du moteur. Enfin, il a aussi dû rédiger la documentation car ayant développé la majeur partie du moteur, il était le plus à même de le faire. Cela explique en partie le retard dans le rendu du projet, nous nous en excusons.

6.2 Suivit du projet

Le suivi du projet a été réalisé en communiquant à l'université mais aussi par le biais d'un groupe Messenger créé spécialement pour le projet de GLP. De plus, nous avons créé un DropBox pour y déposer les versions du programme que nous avons chacun développé en listant les points qui ont été améliorés depuis la version précédente. Des points d'avancement ont été fait au sein du groupe pour connaître régulièrement les tâches restantes.

Temps de travail Tout au long du module GLP, nous nous sommes retrouvés pendant les TD mais aussi pendant les TD en autonomie et parfois même en dehors des cours pour du travail en groupe. Nous avons aussi utilisé un certain temps chacun de notre coté pour développer nos parties respectives, notemment les weekends.

7 Conclusion

Notre projet était donc de développer un jeu de XiangQi, la version chinoise des échecs à l'aide du langage Java. Notre projet comportait un cahier des charges ambitieux comportant notemment une IHM graphique, un jeu qui évolue en fonction de l'environnement ainsi qu'une IA. Nous estimons avoir mené ce projet à bien même si certains points restent à améliorer comme l'IA ou les différents plateaux de jeu. De plus, le jeu comporte toujours quelques bugs à cette date notemment en mode joueur contre IA que nous n'avons pas encore eu le temps de corriger. Nous sommes néanmoins fiers de ce résultat qui nous semble convaincant et à la hauteur de nos ambitions. Un autre point qui aurait été à améliorer est la gestion de projet. En effet comme il a été dit dans la partie 6, la répartition des tâches et des moyens a été très inégale entre le groupe chargé de l'IHM et le développeur du moteur, c'est une faute de groupe. Nous aurions dû mieux répartir les tâches en évaluant précisément la charge de travail au départ sans se fier uniquement aux compétences de chacun.

Enfin, bien que ce projet ait été difficile et long, nous avons appris beaucoup, d'une part grâce au module GLP et d'autre part grace à l'expérience de développement et de recherches personnelles qui ont été effectuées.

Encore merci à M. Liu pour son formidable module GLP et ses cours : CM, TD, programmes d'exemple et présentations. Merci aussi à M. Lemaire qui a animé les cours de POO au premier semestre et qui nous a enseigné les bases du language Java.