Rapport Laboratoire 08
POO 22-23

« Chess »

Ylli Fazlija
Kylian Manzini
Rui Manuel Mota Carneiro

Source : https://upload.wikimedia.org/wikipedia/commons/6/6f/ChessSet.jpg

Table des matières

Introduction

Ce rapport décrit la réalisation du laboratoire 08 du module de Programmation Orientée Objet (POO) de la HEIG.

Membres de l'équipe de réalisation : Ylli Fazlija, Kylian Manzini, Rui Manuel Mota Carneiro

Description du laboratoire

Le but principal de ce laboratoire est de créer un jeu d'échecs. Une interface graphique basique est fournie. La majorité du travail réside dans le fait d'implémenter le déplacement des pièces, l'attaque ainsi que toutes les différentes règles qui gouvernent une partie d'échecs.

Langage de programmation utilisé : Java 17

IDE : JetBrains IntelliJ 2022.2.2

Lien du repository GitHub : https://github.com/FazlijaYlli/LAB8_CHESS

Conception

Schéma UML

Voir annexe à la fin de ce rapport.

Choix de modélisation

Pièces de l'échiquier

Nous avons choisi de représenter les pièces avec plusieurs classes qui possèdent des propriétés différentes. Certaines pièces héritent de classes abstraites tel que CountingMovePiece qui permet de connaître le nombre de mouvements d'une pièce. Toutes ces classes héritent d'une classe abstraite Piece qui regroupe les méthodes générales.

Mouvements et attaques relatives

Dans notre implémentation, les pièces ne connaissent pas leurs positions. Cependant, les pièces peuvent connaître quels sont leurs mouvements relatifs possibles grâce à une méthode spécifique à leur type. La classe Controller peut ensuite utiliser cette information afin de savoir si le mouvement est légal pour cette pièce.

Controller

La classe Controller est le cerveau de notre jeu d'échecs. Il va contrôler toutes les facettes du jeu, comme vérifier les conditions de victoire ou d'arrêt de jeu ou bien permettre un mouvement en contrôlant les différentes conditions préalables. La boucle de jeu se situe dans cet objet. Le contrôleur fonctionne en communication avec la vue afin de transmettre l'affichage du jeu à l'utilisateur.

Réalisation

Nouvelle partie et emplacement des pièces

Il est important de s'assurer de réinitialiser toutes les variables générales de la classe *Controller* afin que l'on puisse appuyer à nouveau sur le bouton "New Game" sans devoir relancer l'application.
Bien que cela puisse paraître illisible, il a été décidé d'effectuer des boucles et des conditions afin de n'utiliser qu'une seule fois le constructeur de chaque pièce.

Mouvements des pièces

Afin de faciliter la création et la modification des pièces ainsi que rendre cela facilement utilisable hors d'un tableau d'échec standard, chaque pièce possède une fonction *canMove()* ainsi que *canAttack()* prenant des paramètres X et Y relatifs à la position de la pièce permettant ainsi à chaque pièce d'indiquer si elle peut se déplacer/attaquer la destination fournie en paramètre. En général, la fonction *canAttack()* appelle la fonction *canMove()*.
Ces fonctions ne servent qu'à vérifier la validité d'un mouvement et toutes les autres vérifications ainsi que mouvements spéciaux sont vérifiés dans la fonction *move()* de la classe *Controller*.

Afin de rendre l'application évolutive et plus simple à coder, chaque mouvement effectue une simulation où l'on bouge chaque pièce et on vérifie si le Roi est attaqué afin de savoir si le mouvement est illégal ou non.

Approche mathématique

Plutôt que de simplement lister des mouvements possibles afin de vérifier la validité d'un mouvement, nous avons trouvé beaucoup plus efficace de vérifier des liens mathématiques entre les valeurs de X et Y, notamment à l'aide de valeurs absolues, additions, multiplications et divisions.

Pion

Le pion est la seule pièce possédant actuellement une fonction *canAttack()* différente. Que ce soit pour attaquer ou se déplacer, le pion, ne pouvant pas reculer, se base sur sa couleur afin de valider la valeur de Y, puis la valeur de X est vérifiée.
Il est également validé de se déplacer de 2 cases en avant tant que le nombre de mouvements est nul. Cela implique que le pion se trouve sur la ligne 7 ou 2, dans quel cas le mouvement de 2 est autorisé.

Tour

La tour nécessitant de se déplacer d'une infinité de cases verticalement ou horizontalement, il est possible de vérifier cela à l'aide d'une addition et multiplication de X et Y.

Chevalier

Le cheval ayant toujours une valeur à 2 et une à 1, cela simplifie bien les choses car on peut donc vérifier cela à l'aide d'une valeur absolue et d'une multiplication.

Fou

Le fou se déplaçant d'une infinité de cases en diagonale, il suffit de vérifier que les valeurs absolues de X et Y soient égales.

Reine

La reine effectue simplement les mouvements de la tour ou du fou.

Roi

Le Roi doit simplement avoir une des deux valeurs à 1 et s'assurer que l'autre valeur ne soit pas exagérée.

Mouvements Spéciaux

Les mouvements spéciaux, comprenant le "Roque" et "En Passant", sont vérifiés et effectués par des fonctions de la classe *Controller* qui sont appelées dans la fonction *move()*.

Roque (Castling)

Le roque se base sur plusieurs critères, que ce soit la vérification de cases vides entre le Roi et la tour, le fait que ces cases ne soient pas actuellement attaquées ainsi que le fait que le Roi et la tour ne doivent pas avoir bougé. La plupart de cette logique est gérée directement par le contrôleur à l'aide d'un tableau de pièces qu'il possède. Le Roi et la tour fournissent des informations sur le fait qu'ils ont déjà bougé ou non via la fonction qu'ils héritent de la classe *CastlingPiece*.

Prise En Passant

"En Passant" est le mouvement le plus compliqué des échecs car il nécessite des conditions très précises afin de pouvoir se réaliser.
Afin de pouvoir savoir si ce mouvement est éligible, nous utilisons une fonction héritée de la classe *CoutingMovePiece* permettant de savoir la nombre de mouvements que la pièce a effectué.
Toutes les autres vérifications sont faites par le contrôleur.

Promotion du Pion

Dans la fonction *endOfTurn()* de la classe *Controller* qui est appelée après qu'un mouvement ait été effectué, si un pion a atteint la dernière ligne du plateau, on appelle la fonction *promotion()* de la classe *Controller* qui va demander à l'utilisateur quelle pièce il souhaite obtenir. Cela est effectué via les outils qui nous sont fournis.

Mise en échec "Check"

Une fonction *isCellAttacked()* prenant en paramètres la couleur de l'attaquant ainsi que la position de l'attaque a été mise en place afin de facilement vérifier si une position est attaquée. Cela permet de facilement vérifier si un "Check" a lieu en demandant si la cellule du Roi est attaquée. La fonction *currentPlayerKingPos()* permet de récupérer facilement la position actuelle du Roi de la bonne couleur.

Le booléen "Check" sert uniquement à afficher le message "Check" et à détecter l'échec et mat. Il n'est pas utilisé pour les mouvements du prochain tour car on vérifie toujours que le Roi ne soit pas en échec après un mouvement.

Partie nulle

A la fin de la fonction *endOfTurn()* s'effectuent des vérifications de fin de partie, notamment les "Stalemates".

Ici, il y a 3 booléens "stalemateByMoves", "stalemateByPat" et "checkmate" qui permettent de savoir ce qu'il s'est passé afin de changer l'affichage de messages ainsi que terminer la partie.

Une fois la partie terminée, le booléen "endOfGame" est mis à vrai et la seule action possible est de recommencer une partie.

Par "Stalemate"

Le "Stalemate" standard a lieu quand un joueur ne peut plus effectuer de mouvement. Pour cela, pour chaque pièce de la couleur du joueur, on effectue des simulations où on tente d'attaquer toutes les cases et on vérifie si le Roi est attaqué via la fonction *isCellAttacked()*.

Par "Pat"

Le "Stalemate by Pat" est plus spécifique car il représente les occurrences où il n'est plus possible de mettre "Echec et Mat". Celles-ci représentent :

- Roi contre Roi
- Roi contre Roi et Fou
- Roi contre Roi et Chevalier
- Roi et Fou contre Roi et Fou (à condition que les deux fous soient sur des cases de couleur identiques)

Echec et mat "Checkmate"

Echec et mat est assez simple car il reprend simplement le booléen "stalemateByMoves" et le booléen indiquant un "Check" pour faire un "Echec et Mat".

Tests

COMPORTEMENT RECHERCHÉ	
Initialisation	👍
Appuyer sur New game affiche les pièces	👍
Les pièces sont affiché aux bons endroits	👍
Mouvement et attaques de base	👍
Les mouvements de base du pion fonctionnent	👍
Les mouvements de base du roi fonctionnent	👍
Les mouvements de base de la reine fonctionnent	👍
Les mouvements de base du fou fonctionnent	👍
Les mouvements de base du cheval fonctionnent	👍
Les mouvements de base de la tour fonctionnent	👍
Les attaques de base du pion fonctionnent	👍
Les attaques de base du roi fonctionnent	👍
Les attaques de base de la reine fonctionnent	👍
Les attaques de base du fou fonctionnent	👍
Les attaques de base du cheval fonctionnent	👍
Les attaques de base de la tour fonctionnent	👍
En passant	👍
Fonctionne uniquement quand le dernier mouvement est un pion qui a avancé de 2	👍
Ne fonctionne pas si le pion ne se trouve pas sur la case adjacente	👍
Ne fonctionne pas si le pion est de la même couleur	👍
Ne fonctionne pas si la pièce adverse n’est pas un pion	👍
Ne fonctionne pas si la prise “En passant” met en échec son propre roi	👍
“En passant” fonctionne pour les noirs	👍
“En passant” fonctionne pour les blancs	👍
Roque (Castling)	👍

COMPORTEMENT RECHERCHÉ	
Ne fonctionne pas si une des cases entre le roi compris à la tour non comprise est attaquée	👍
Ne fonctionne pas si une des cases entre le roi et la tour est occupée	👍
Ne fonctionne pas si le roi a déjà bougé	👍
Ne fonctionne pas si la tour a déjà bougé	👍
Grand roque noir fonctionnel	👍
Petit roque noir fonctionnel	👍
Grand roque blanc fonctionnel	👍
Petit roque blanc fonctionnel	👍
Mise en échec - Check	👍
Lorsque le roi est attaqué, un échec doit être pris en compte	👍
Affichage de “Check !” sur la vue	👍
Le roi ne peut pas se déplacer sur une case attaquée	👍
Le roi attaqué peut se déplacer pour contrer la mise en échec	👍
Le roi attaqué peut bloquer la mise en échec avec une autre pièce, si la mise en échec ne vient pas d’un cheval	👍
Le roi attaqué peut manger la pièce attaquante pour contrer la mise en échec	👍
Le roi attaqué peut manger la pièce attaquante avec une autre pièce pour contrer la mise en échec	👍
Partie nulle - Stalemate	👍
La partie est nulle lorsque qu’il n’existe plus de coups possible sans que le roi soit en échec	👍
Partie nulle - Stalemate par Pat	👍
La partie est nulle s’il reste roi et roi	👍
La partie est nulle s’il reste roi et roi et cheval	👍
La partie est nulle s’il reste roi et roi et fou	👍
La partie est nulle s’il reste roi, fou et roi, fou avec les fous sur des cases de même couleur	👍

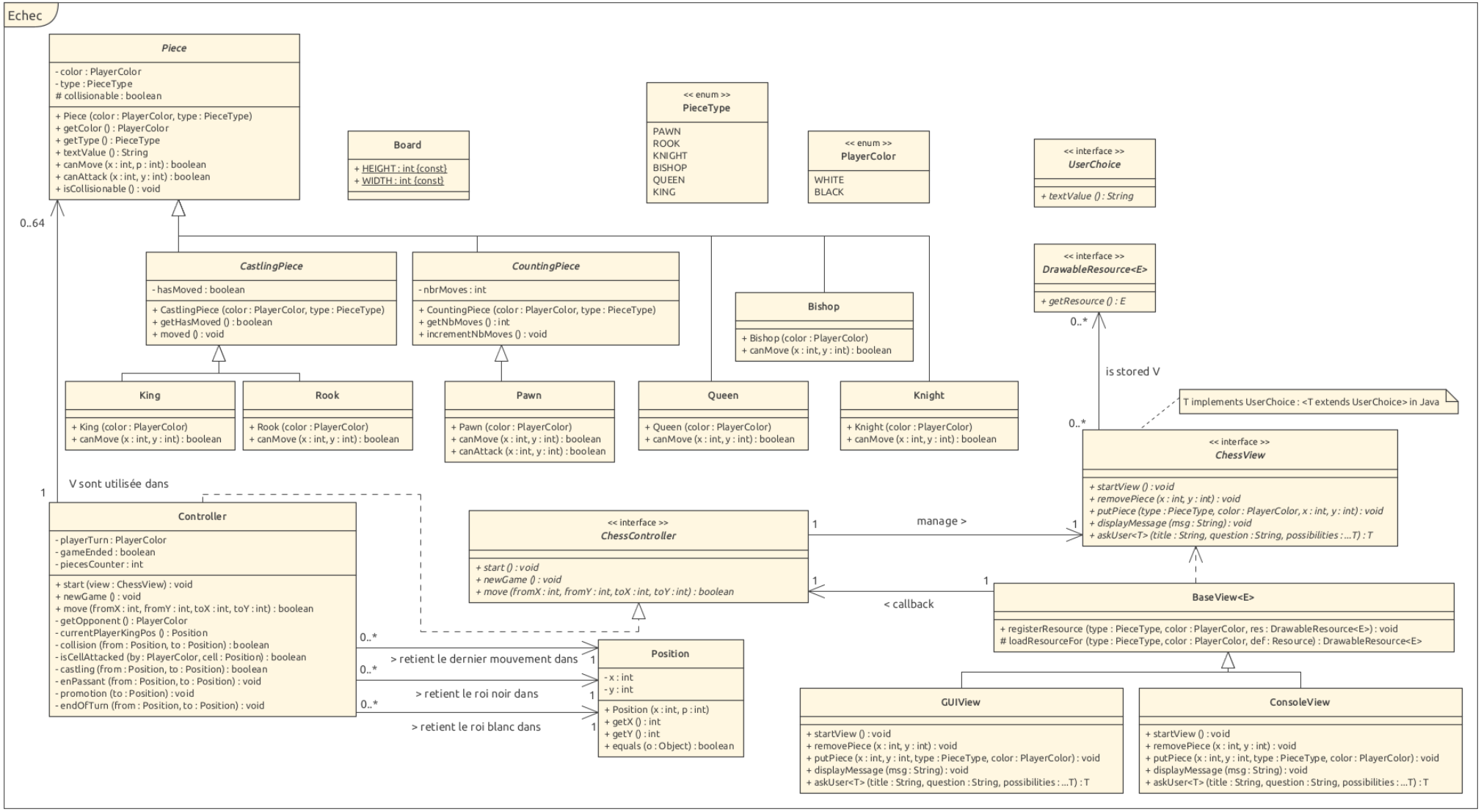
COMPORTEMENT RECHERCHÉ	
Affichage de “Stalemate !” dans la vue	👍
Echec et mat - Checkmate	👍
La partie est gagnée lorsque qu’il n’existe plus de coups possible et que le roi est en échec	👍

Conclusion

La réalisation de ce laboratoire nous a permis d'appliquer les compétences acquises durant les cours ainsi que les précédents laboratoires de POO. En outre, nous avons essayé d'utiliser un maximum le feedback des derniers laboratoires rendus afin de peaufiner le rendu de celui-ci.

La réalisation d'un laboratoire en équipe de trois nous a aussi permis de mettre en avant les outils de développement en équipe comme git. Au lieu de simplement créer des branches portant nos noms chacun, nous avons plutôt créé une branche par feature que nous étions en train de développer.

Nous avons aussi utilisé un outil de type Kanban afin de nous faciliter l'organisation générale du projet. Cela s'est révélé très utile par la suite car nous n'avons pas eu besoin de nous concerter à chaque fois qu'une personne avait besoin d'une nouvelle tâche.



Annexe 1 : Schéma UML