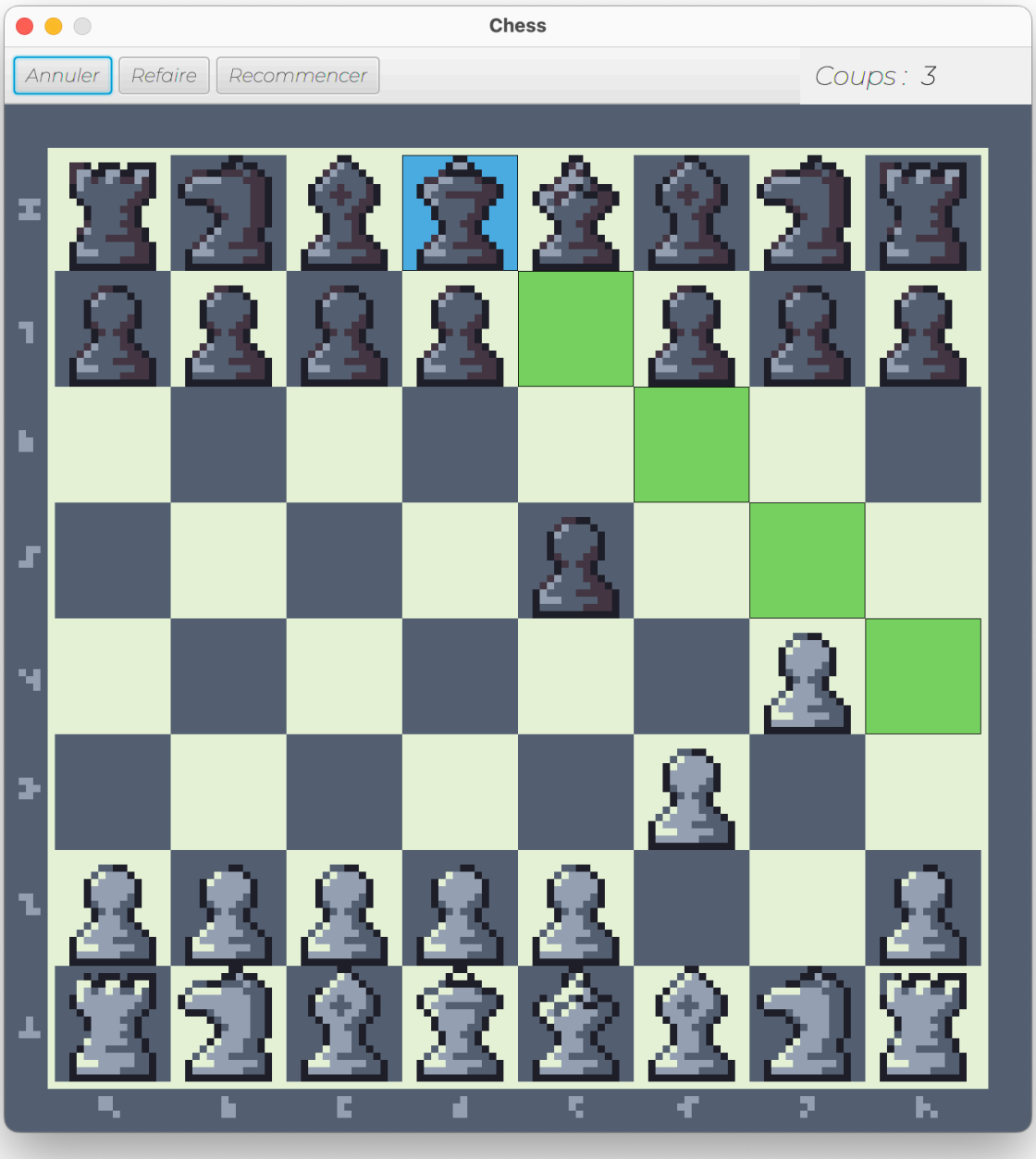

Rapport Projet POO2

Jeu d'Échec

Efe ERKEN - L2S4P - TD2-TP4

En tant qu'étudiants en licence informatique, dans le cadre de l'UE Programmation Orienté Objet 2, nous avons du réaliser un projet de fin de semestre. Le sujet était de concevoir et d'implémenter un jeu d'échec et un variant de celui-ci avec une interface graphique écrit dans le langage de programmation Java. Voici un compte rendu qui explique mes choix d'implémentation, mon interprétation des demandes du sujet, ainsi que le fonctionnement sous le capot de mon programme. Ce document n'explique pas comment démarrer et utiliser l'exécutable final, pour cela referez vous au fichier "README.md" dans le code source.

J'ai réalisé ce projet seul pourtant nous avions le droit de le faire en binôme si on le souhaitait. Le sujet indiquais que les étudiants travaillant seul n'étaient pas obligés de satisfaire certaines demandes du sujet comme par exemple les règles du jeu variant. Pour cette raison, ma production finale ne correspond pas à 100% aux demandes dans le sujet et c'est normale.

Je vais parler brièvement de mon avis personnelle sur le projet. La tâche qui m'a été donné par ce projet était un exercice profond et détaillé qui m'a beaucoup aidé à mieux comprendre le concept de la POO, ainsi que la manière de penser/l'idéologie de celle-ci. Elle m'a aidé à mettre en pratique et à concrétiser les concepts et les connaissances acquis lors des cours théoriques comme dans les CM. Elle m'a aidé à prendre du pratique et expérimenter à l'aide de Java la POO. En parlant de Java, le coté relativement haut niveau de ce langage a beaucoup aidé dans ce projet pour l'implémentation de chaque fonctionnalité. Au lieu de me battre avec le langage, j'ai pu avancer dans mon programme et me concentrer sur mes algorithmes contrairement à l'UE Architecture. Pourtant j'aimerais dire que la tâche du sujet était très large avec beaucoup de demandes sous entendu comme tous ce qu'il faut réussir à faire pour concevoir un jeu d'échec fonctionnel sans bogues. Ce n'est pas facile mais surtout prend beaucoup de temps (un vrai problème algorithmique auquel il faut beaucoup réfléchir). Juste dire qu'il faut faire un jeu d'échec a l'air simple et facile mais en fait il y a beaucoup derrière. Tout cela pour dire qu'en fait la tâche donnée était excellent mais le temps qu'on avait pour l'accomplir était très limité en vue de tous les examens et d'autres UE qu'on avait. C'est pour cela il y reste certaines fonctionnalités et demandes du sujets que je n'ai pas réussi à compléter. La solution à cela à mon avis n'est pas de publier le sujet plus tôt dans le semestre car pour le commencer il faut avancer à un certain niveau dans la matière, du coup il faut peut-être raccourcir les demandes du sujet ou assouplir la notation.

Bref, j'aimerais passer au vif du sujet de ce rapport. Je vais commencer par les fonctionnalités réussis ou non de ma production finale. Puis, je vais parler plus en détails de mes choix de modélisation et d'implémentation où je parle aussi des algorithmes que j'ai développé. À la fin, le diagramme de classes simplifiés va vous aider à visualiser le structure interne du programme.

Fonctionnalités et Demandes Réussis/Non Réussis

J'ai réussi à faire un jeu d'échec standard complet où une partie complète peut être joué sans problèmes avec bien sûr les fonctionnalités de bases suivantes :

- Le calcul des mouvements possibles pour chaque pièce pour mettre ces cases en évidence
- Le mouvement de chaque pièce standard (aucun coup illégal n'est permis ou n'est laissé faire comme dans un vrai jeu d'échec)
- La prise/capture de pièce
- Le calcul du fin du jeu (Echec et mat ou pat)
- Les coups plus complexes comme (le roque, la prise en passant et la promotion de pion)

Sauf pour les fonctionnalités :

- Les pièces variantes/féériques
- Les règles de variant

Puis, au niveau application et non pas algorithmique, j'ai réussi à faire une interface graphique en JavaFX avec le patron de conception MVC comme on l'a vu en cours et en TD. Il est possible de recommencer une nouvelle partie à la fin du jeu si vous voulez. Le code source est fourni et est organisé et compilable ainsi que le fichier unique ".jar" selon votre système d'exploitation qui est normalement possible d'être lancé par un simple double clique.

Finalement, ce que je voulais faire qui n'était pas forcement demandé dans le sujet et pour lesquelles je n'ai pas eu le temps de finir. Je voulais documenter tout mon code avec javadoc. Je n'ai pas eu le temps de mieux organiser mon code, optimiser l'encapsulation et optimiser les algorithmes et le fonctionnement elle-même. Eh oui, ma production finale n'est pas bien optimisé même si elle marche bien.

Choix de Modélisation, d'Implémentation et les Algorithmes

J'aimerais commencer du composant le plus petit et avancer vers les plus haut niveaux qui sont le jeu et son contrôleur. Je vais utiliser les termes position et coordonnées de manière interchangeable dans la suite. Entre parenthèses et des guillemets sont les noms des classes.

Coordonnées ("Coordinates")

J'ai crée une classe pour représenter les coordonnées sur un plan au sein du package "model". Cette classe contient deux entiers pour représenter l'abscisse et l'ordonnée. Je l'ai utilisé comme une "struct" en C pour organiser le code (pour raccourcir et simplifier les paramètres des méthodes et pour améliorer la lisibilité).

Pièces ("Piece", "RegularPiece" & "RoyalPiece)

J'ai créé une classe abstraite "Piece" pour rassembler les points communs des pièces dans le jeu. Une pièce a une couleur telle que noir ou blanc, elle a une liste de coordonnées des positions légales et une autre liste pour les positions auxquelles elle attaque. Elle a aussi un indicateur pour dire si elle est en train de protéger une pièce royale. Si oui elle a aussi la référence de la pièce qui la cause à protéger sa pièce royale.

À part ses accessseurs et son constructeur, elle fourni les 3 méthodes par défaut "coordinateCheck" pour vérifier si les coordonnées données se trouvent bien à l'intérieur du plateau de jeu et "updateAllPositions" qui fait recalculer toutes les positions légaux et d'attaques de la pièce en vue de la situation actuelle du plateau de jeu, et enfin "destinationPieceCheck" qui vérifie si la pièce à la case destination est bien inexistante ou de la couleur opposé.

Elle définit des méthodes abstraites pour vérifier si une position est légal ou est une position d'attaque pour elle ainsi que la vérification d'une position si elle est dans le chemin de la pièce qui va jusqu'au roi opposé (utilisé pour éviter des coups illégaux qui laissent le roi en danger).

J'ai étendue cette classe abstraite avec deux classes abstraites encore pour représenter les pièces normales et royales. Une pièce royale peut être en échec et avoir des attaquants. Elle a un indicateur pour dire s'il est en échec et si oui il a aussi une liste de pièces qui le mettent en échec. Une pièce normale peut mettre une pièce royale en échec et peut produire des protecteurs royaux. Il y a "setOppositeKingToCheck" pour mettre le roi de la couleur opposé en échec s'il est en danger par cette pièce et le marquage des protecteurs de roi de la couleur opposée.

J'ai étendue ces classes abstraites avec des classes concrètes finales pour représenter les 6 vraies pièces du jeu. Les classe concrètes des pièces ont chacun leur propres surdefinitions et méthodes privées d'aide pour leur cas spécial.

Chaque pièce a une méthode privée qui traverse tout son chemin et utilise sa méthode de vérification si la position concerné est valide. Pour simuler les pointeurs de fonction de C j'ai créé une interface générique "Predicate3" et "Predicate4". J'ai fait cela pour changer la méthode de vérification utilisé dans la méthode de parcours de positions selon les cas. Ensuite, ces positions renvoyé passent sous une deuxième vérification selon la priorité de danger (roi en échec, protecteur de roi, mode normal), puis les positions valides sont ajoutés aux listes correspondantes. Après avoir recalculé chaque pièce met en échec le roi opposé si possible et met en état de protecteur de roi les pièces opposé correspondantes si possible encore. Si son roi est en échec, une pièce ne peut bouger dans une position que si cette dernière sauve son roi. Si une pièce est en état de protecteur de roi, elle ne peut bouger dans une position que si cette dernière ne laisse pas son roi en danger. Sinon, la pièce bouge là où elle veut tant que ses conditions sont satisfaits comme par exemple la destination n'est pas bloqué par une autre pièce sauf si la pièce est une pièce qui saute (comme le cavalier) ou bien si la destination n'a pas une pièce de la même couleur qu'elle.

Je vais mentionner les pièces qui ont des champs ou des méthodes en plus.

Le pion ("Pawn")

Il a un champ en plus pour indiquer son premier coup où il a le droit d'avancer de deux cases. Il a une méthode de vérification en plus pour déterminer le coup "en passant".

Le roi ("King")

Dans son mouvement il a deux conditions de plus qui compliquent sa réalisation. Il ne peut bouger dans une position que si cette position n'est pas en état de danger relative à sa couleur (car sinon, au prochain coup le roi est pris ce qui est illégal comme coup). Il ne peut non plus bouger nulle part dans le chemin d'une pièce qui le met en échec, il doit dégager ou être sauvé si possible par une autre pièce de sa couleur.

Pour ces vérifications je me suis rendu compte à un moment avancé dans le projet, qu'il fallait mettre en place un système de calcul et de sauvegarde de danger des cases pour chaque couleur ainsi que des vérifications pour savoir si un coup d'une pièce peut sauver le roi ou non.

Cases du plateau ("Square")

La classe "Square" contient une référence vers la pièce que la case contient ainsi que deux indicateurs : "SquareDanger" qui est une énumération pour dire qu'elle est attaqué par quelle couleur et "SquareState" pour représenter la sélection d'un joueur d'une case et pour savoir si la case est mis en évidence (utilisé pour indiquer une position légal).

Plateau de jeu ("Board")

Les vraies positions des pièces et des cases commencent ici. Ni les pièces, ni les cases stockent une information sur leur coordonnées, elles ne le savent pas. C'est le plateau du jeu qui met cela en place.

Il a deux constantes pour indiquer le largeur et la longueur maximale du plateau. Il a un tableau à deux dimensions de cette taille de type "Square" (Case) qui est la grille de jeu. Et finalement, il a deux références vers les deux rois dans le plateau pour faciliter et optimiser leur recherche. Quand le plateau est créé, il rempli la grille avec la configuration standard des pièces (blancs en bas, noirs en haut). Il a les méthodes nécessaires pour retrouver à partir d'une pièce ses coordonnées, retrouver une case à partir des coordonnées et vice versa. Il a une méthode pour sélectionner la case d'une piece pour mettre en évidence ses coups légaux. Il a une méthode pour actualiser le danger de tous les cases. Il peut effacer tout les indicateurs d'état des

cases. Finalement il peut recalculer les positions de toutes les pièces et compter le nombre de tou les coups légaux possibles pour une couleur.

Puisque la grille est une matrice, j'ai beaucoup utilisé deux boucles "for" imbriquées dans les méthodes de cette classe.

Coups ("Move", "RegularMove", "EnPassantMove", "PawnPromotionMove", "CastlingMove", "MoveFactory" & "MoveHistory")

La classe abstraite "Move" possède 2 champs. Le joueur qui a effectué le coup et un indicateur pour dire si le coup a été exécuté. Un coup peut être fait et annulé. Il doit fournir les informations sur la pièce qui a effectué le coup et les coordonnées associé aux cases qui ont subi des modifications (pour mettre à jour juste les parties concernées de l'interface graphique).

J'ai étendu cette classe abstraite par 4 classes concrètes finales pour représenter les 4 coups différents dans le jeu : coup simple, en passant, le roque et la promotion de pion. Elles redéfinissent les méthodes d'exécution de coups avec leur propre implémentation. Avant tout elles vérifient bien sûr si le coup est légal en accédant la pièce dans la case source et sinon elles lèvent une exception personnalisée "IllegalMoveException". Le coup pour promouvoir un pion lève l'exception "PawnPromotionException" si un choix de pièce n'est pas effectué ou c'est une mauvaise pièce.

J'ai utilisé le patron de conception "Factory" pour créer ces coups spéciaux. "MoveFactory" détecte de quel coup il s'agit et renvoie le bon coup polymorphe.

J'ai implémenté une fonctionnalité d'historique de coups pour revenir en arrière. Principalement je pensais que j'allais avoir besoin d'un tel système pour le coup "en passant" mais finalement je m'en suis servi pour une fonctionnalité "annuler/refaire" car mon système était devenu trop mature pour n'être utilisé que pour le calcul d'un coup simple.

Joueurs ("Player")

Un joueur a une couleur et une case représentant sa sélection dans le plateau. Il a deux méthodes : une pour faire une sélection dans le plateau et l'enregistrer dans son champ, et une autre pour effectuer un coup où il crée et exécute un coup et efface la sélection déjà fait.

Jeu ("Game")

Cette classe est le coeur du jeu. Elle rassemble toutes les fonctionnalités des classes plus petites pour former la boucle et la logique de jeu. Il a un plateau de jeu, deux joueurs de couleurs opposées, une historique de coups (non implémenté), une référence vers le joueur du tour actuel et un indicateur sur la raison de fin du jeu (Echec et mat ou pat ou rien). Il joue le rôle d'interface entre le model et le monde extérieur (dans notre cas le contrôleur). Le monde extérieur interagit avec elle pour faire des coups, faire des sélections et savoir la raison de fin de jeu. Tout autre fonctionnement est interne et est encapsulé.

La boucle de jeu est le suivant :

- Coup effectué
- Sélection et mise en évidence des cases est effacé
- Danger des cases est effacé
- Les rois sont remis à zéro (Effacer l'état en échec et les pièces attaquantes)
- Les protecteurs de roi sont remis à zéro (Effacer l'état protecteur et la pièce attaquante)
- Toutes les pièces de la même couleur que le joueur qui a fait le dernier coup recalculent leurs positions

- Le danger des cases est actualisé
- Toutes les pièces de la couleur opposée recalculent leurs positions
- Le danger des cases est actualisé
- Le tour est avancé (il passe à l'autre joueur)
- Vérifications du fin du jeu
    - Si le roi du joueur actuel est en échec et il n'y a aucun coup possible -> Echec et mat (Fin du jeu)
    - Si le roi du joueur actuel n'est pas en échec mais il n'y a aucun coup possible -> Pat (Fin du jeu)
- Faire un nouveau coup

Si le jeu est terminé mais on essaye de faire des coups ou des sélections, l'exception personnalisée "EndOfGameException" est levé et aucune autre opération est effectué (le jeu est en état de lecture seul). Si le bouton "annuler" est appuyé, le dernier coup est reversé et tout est recalculé, même chose avec le bouton "refaire" mais vers le futur. S'il n'y a pas de passé ou de futur rien n'est effectué lors de l'appui des boutons. Le bouton "Recommencer" est pour commencer une nouvelle partie à tout moment du début.

Vue ("GUIJavaFX")

Cette classe qui étend la classe "Application" de JavaFX charge le fichier FXML, crée le contrôleur du jeu en initialisant ses champs annotés @FXML et dessine la fenêtre principale.

Controleur ("GameController")

Le contrôleur parcours la grille GridPane FXML et charge les éléments nécessaires à manipuler dans ses champs. Le GridPane contient 64 StackPane modélisant chaque case. Les StackPane contiennent un Rectangle et un ImageView chacun. Les cliques sur chaque StackPane déclenche le calcul de ses coordonnées x et y dans le GridPane et soit une sélection de case dans le jeu ou un coup à effectuer.

La classe statique "Constants" est utilisé pour déterminer la palette de couleurs du jeu, les chemins des fichiers ".png" pour les images des pièces correspondantes, ainsi que les chaines de caractères pour traduire en français la raison de fin du jeu. J'ai accompli les associations clé-valeur avec des "Map" Java et en particulier le "HashMap" pour simuler les tuples Python avec recherche par clé.

À chaque clique dans une case StackPane la méthode "selectOrMove" du contrôleur est lancé qui détermine s'il faut faire une sélection de case ou un coup et lance l'opération dans le model. Ces opérations, à la fin, rafraîchissent les cases et les pièce concernés dans la vue. Si l'exception "EndOfGameException" est levé par le jeu, celle-ci est attrapé et la méthode "endGame" est lancé qui affiche une alerte proposant le choix de commencer une nouvelle partie.

Si une sélection est effectué les cases mises en évidence sont affiché avec le rafraichissement. Si l'exception "IllegalMoveException" est levé une petite animation indique que cette case n'est pas permis.

Si un coup est effectué, pour optimiser, au niveau des images des pièces, juste les cases concerné sont rafraichi et les autres ne sont pas touché. Puisque les images utilisé sont de haute qualité, rafraichir tout pour rien introduisait des ralentissement que j'ai évité comme cela.

Les boutons dans l'interface en haut, déclenchent des méthodes dans le contrôleur (annuler, refaire et recommencer). Il y a aussi un indicateur sur le nombre de coups effectué dans la partie qui est mis à jour à chaque coup.

Diagramme de Classes

Le diagramme plus détaillé avec les méthodes et les champs est disponible dans les fichiers du projet si vous voulez y jeter un oeil.