RAPPORT DE PROJET : CHESS4J

Professeur encadrant : MERCADAL Julien

Projet de Groupe :
avec la participation de :
HUANG Pascal, KY Vincent, MOEUNG Alim, NGUYEN-CAO Quoc Thai (James), OULHEN Jules, TRAN Nhat Anh

23/05/2022

Sommaire :

1. INTRODUCTION

Afin de réaliser un jeu d'échecs en Java, tout en respectant le cahier des charges du projet, nous avons dans un premier temps réfléchi à la conception de ce dernier. L'ensemble de cette conception est ainsi reflété par nos diagrammes UML (diagramme de cas d'utilisation, diagramme de classe et diagrammes de séquence). Grâce à ce travail, nous avons ensuite pu commencer l'implémentation du noyau applicatif en Java avec une interface en ligne de commande afin de tester notre code. Enfin, nous avons implémenté une interface graphique pour une utilisation plus simple et agréable, grâce à JavaFX.

Nous allons donc commencer par présenter nos choix de conceptions à travers la présentation de nos diagrammes UML. Ensuite, nous expliciterons les réflexions que nous avons eu, ainsi que les choix que nous avons dû faire, tout au long du processus de création de l'application : de la conception, jusqu'au passage à l'interface graphique, en passant par l'implémentation du noyau applicatif. Enfin, nous finirons par proposer des axes d'amélioration d'un point de vue structurel (implémentations) mais aussi des idées d'ajouts de fonctionnalités non présentes dans le cahier des charges.

2. UML

a) Diagramme de cas d'utilisation

Nous avons commencé à réfléchir à notre sujet en réalisant un diagramme de cas d'utilisation d'un simple joueur (figure n°1). Ce dernier doit être capable avant tout de lancer une partie. Suite à ça, l'utilisateur à la possibilité de voir et choisir les pièces qui sont bougeables. Toutes les pièces qui sont jouables sont mises en surbrillance pour pouvoir aider le joueur. Ensuite il peut en sélectionner une et alors les déplacements possibles de la pièce seront mis en surbrillance. L'action de prendre une pièce adverse consiste à déplacer sa pièce sur une pièce adverse. Alors le jeu met à jour le compteur de pièces prises avec la nouvelle pièce.

Lors de la partie, le joueur peut tourner le plateau de jeu. Il a également la possibilité d'arrêter la partie quand il le souhaite, sous condition qu'une partie soit en cours. Néanmoins, au lieu de simplement quitter, le joueur peut aussi décider de sauvegarder sa partie pour la continuer plus tard. Il pourra ainsi recharger une ancienne partie précédemment sauvegardée.

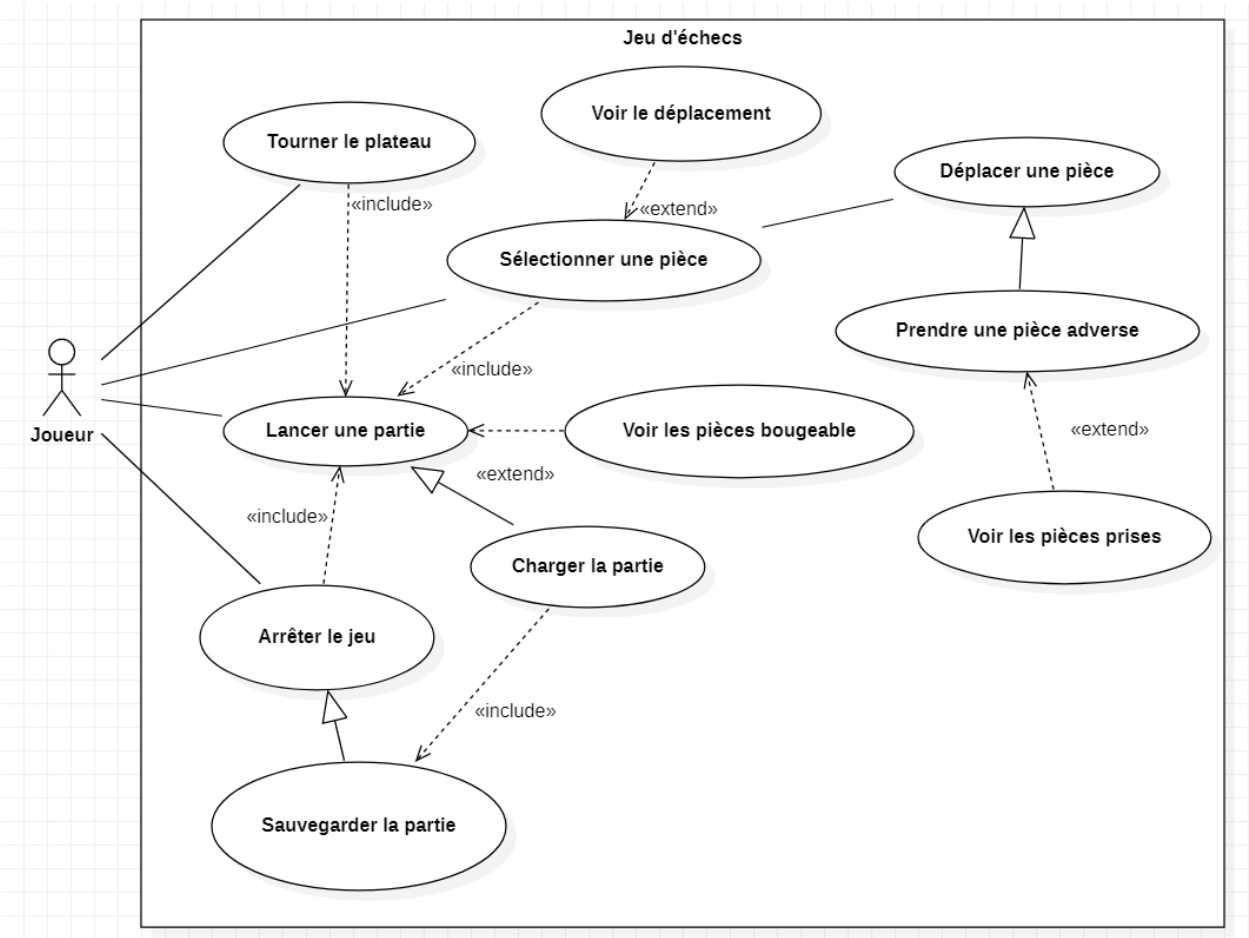


Figure n°1: Diagramme de cas d'utilisation

b) Diagramme de classe

Après avoir réalisé le diagramme de cas d'utilisation, nous avons conçu le diagramme de classe, indispensable pour l'implémentation en Java (figure n°2).

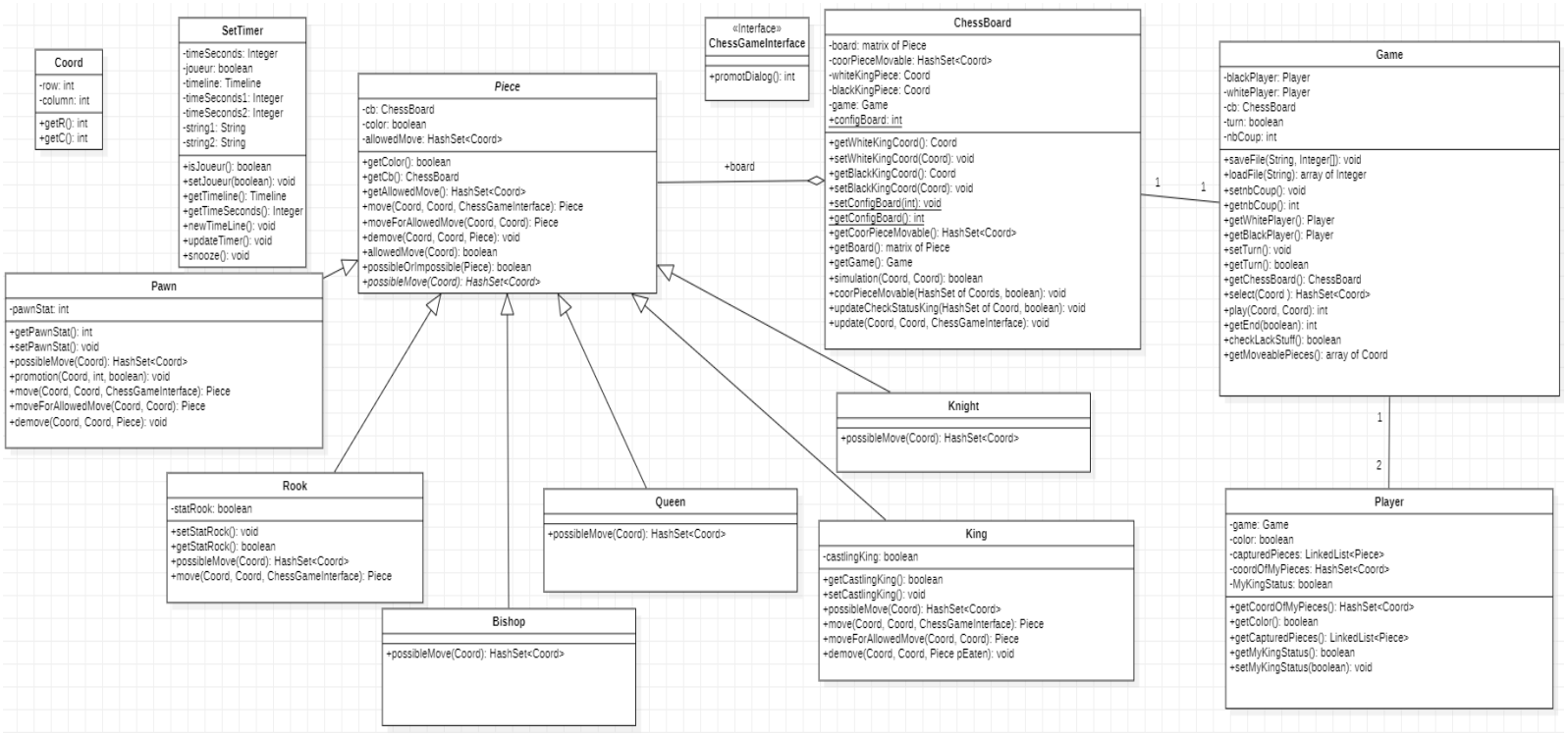
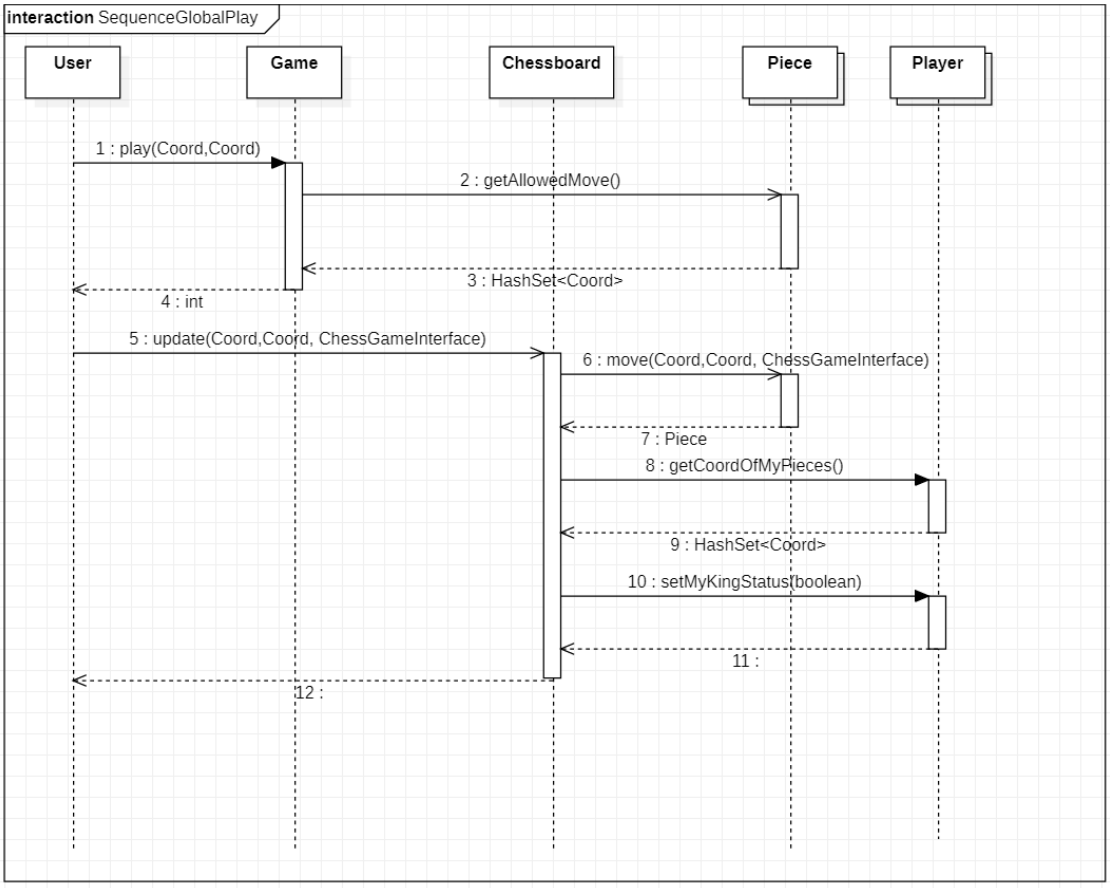


Figure n°2: Diagramme de classe

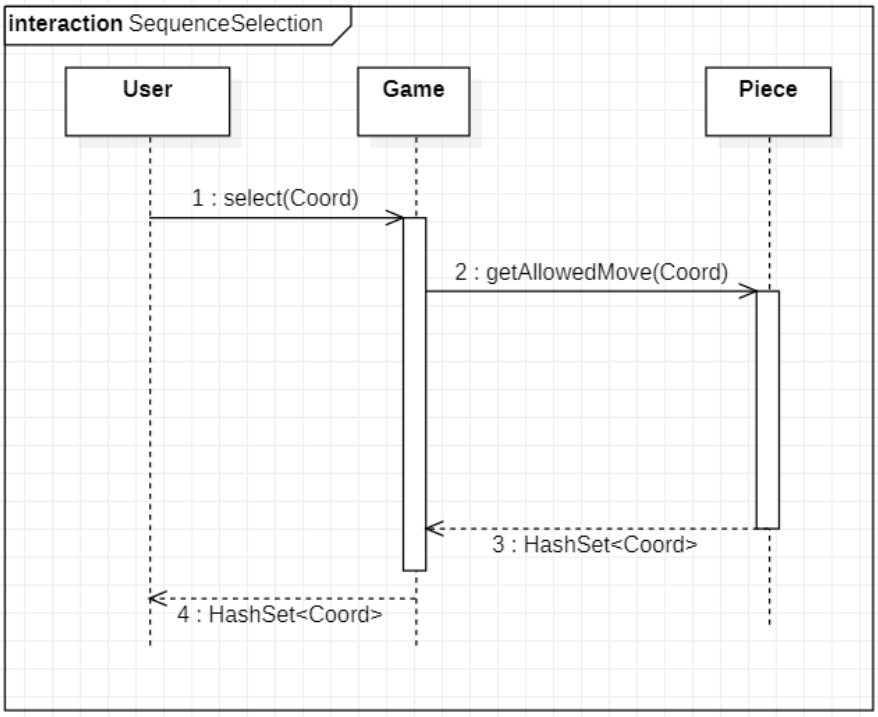
c) Diagramme de séquence

Nous avons conçu sept diagrammes de séquence différents, représentant la plupart des situations se produisant durant le jeu.

- GlobalPlay : Il représente le moment où le joueur souhaite jouer. Il ne représente cependant pas les cas particuliers, notamment concernant la pièce **`Pawn`** ou la pièce **`King`**. Ce dernier cas sera représenté dans le diagramme de séquence *Pawn Promotion*

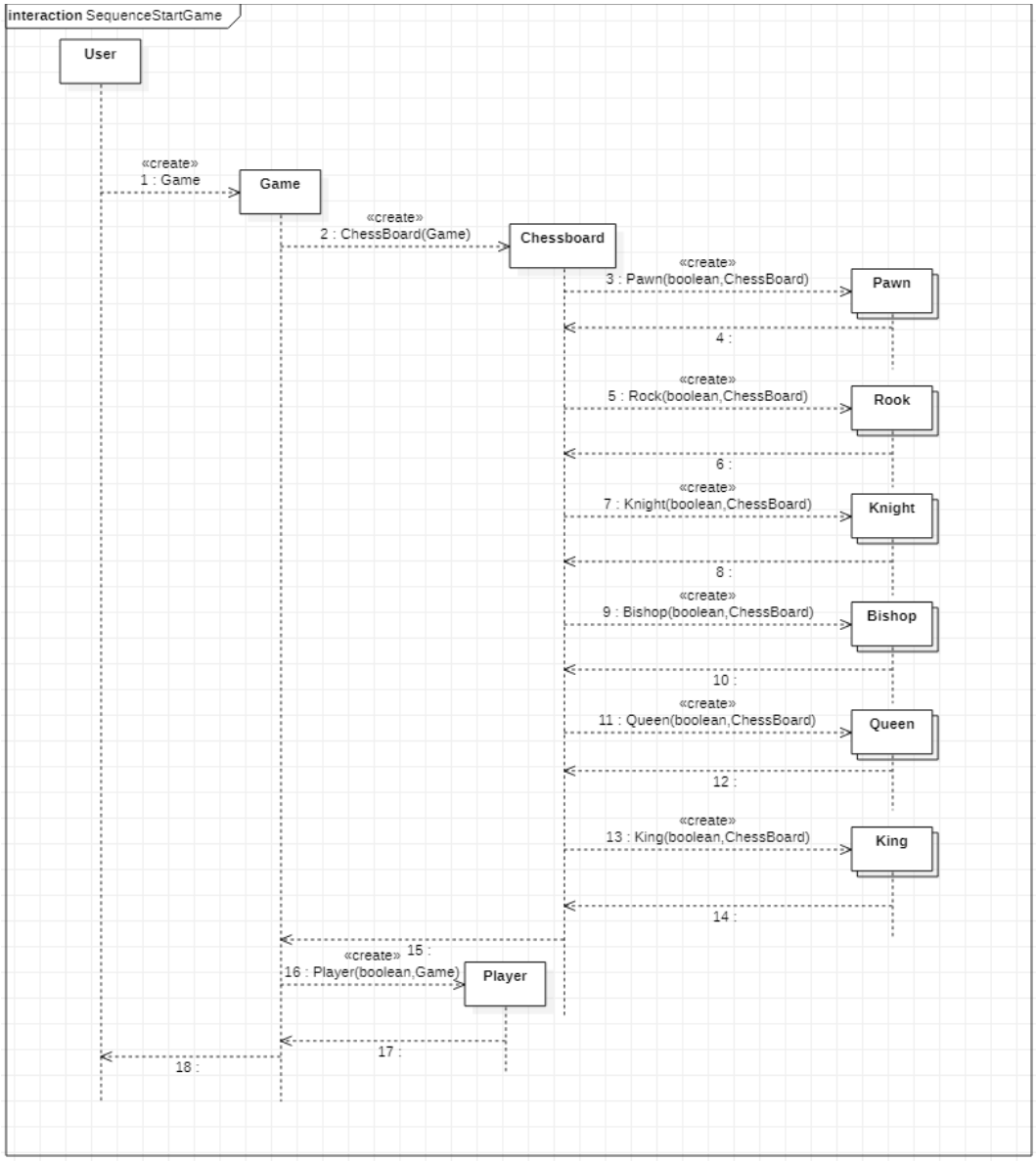


- Selection : Il représente le moment où le joueur souhaite sélectionner une pièce, et ainsi voir les déplacements possibles de cette dernière.

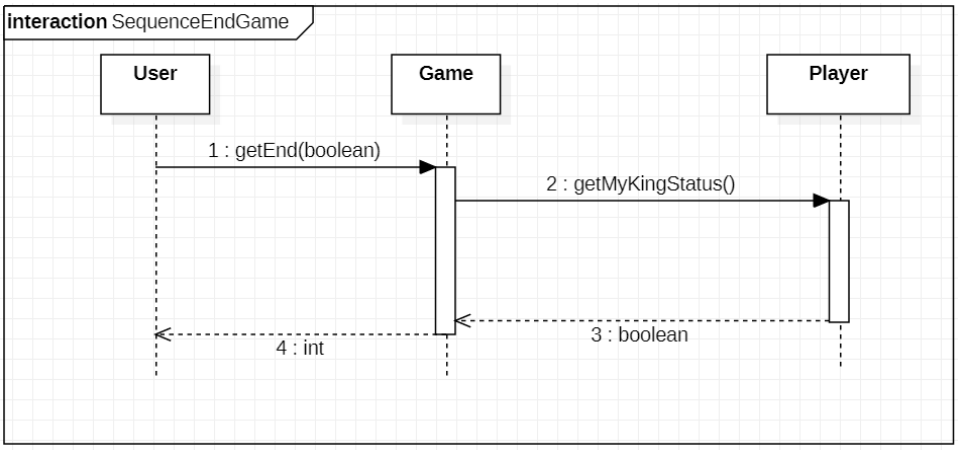


-

- Start Game : Il représente le début de la partie, lors de l'initialisation de chaque pièce, des joueurs et du plateau de jeu.

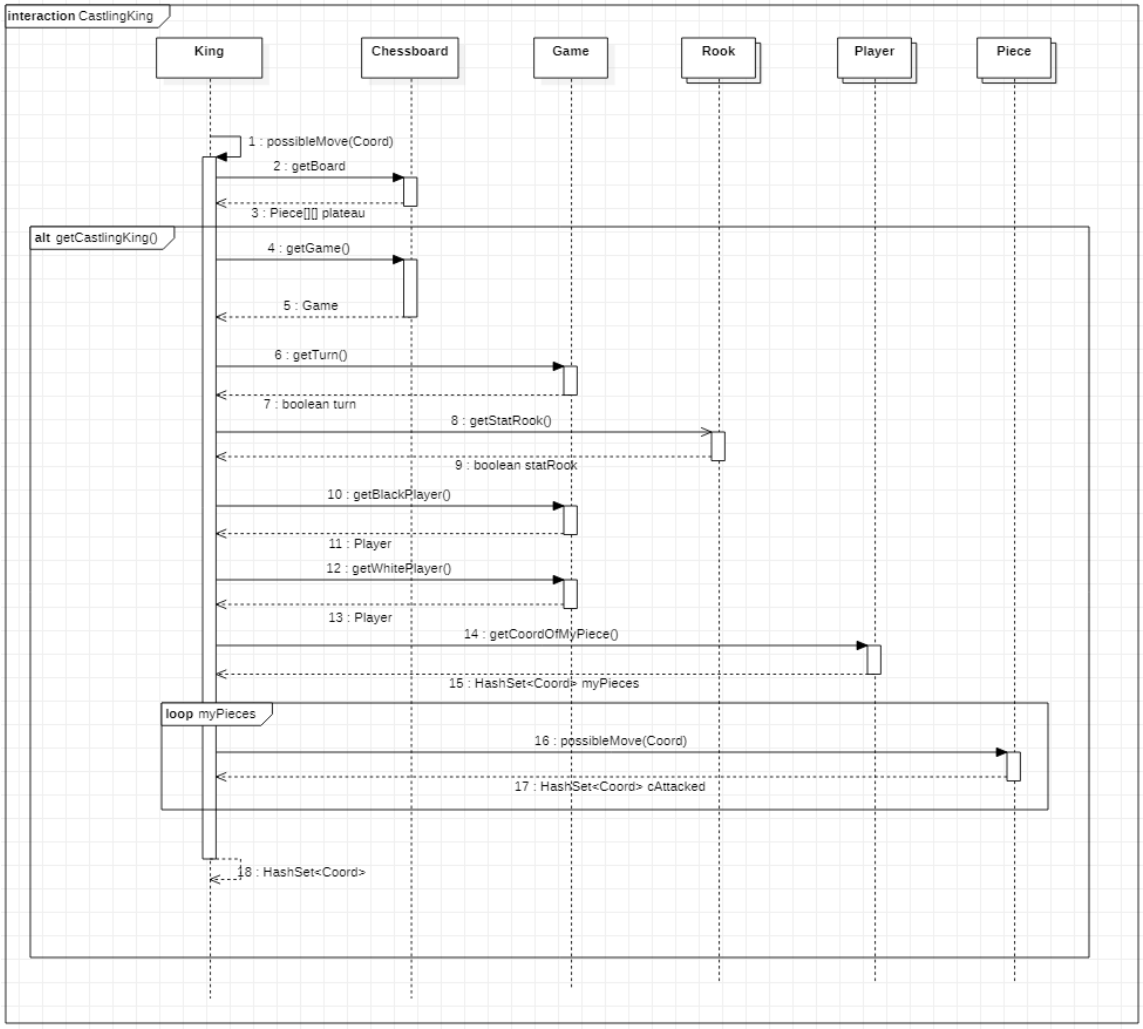


- EndGame : Il représente la fin de la partie, lors de l'annonce du potentiel gagnant.



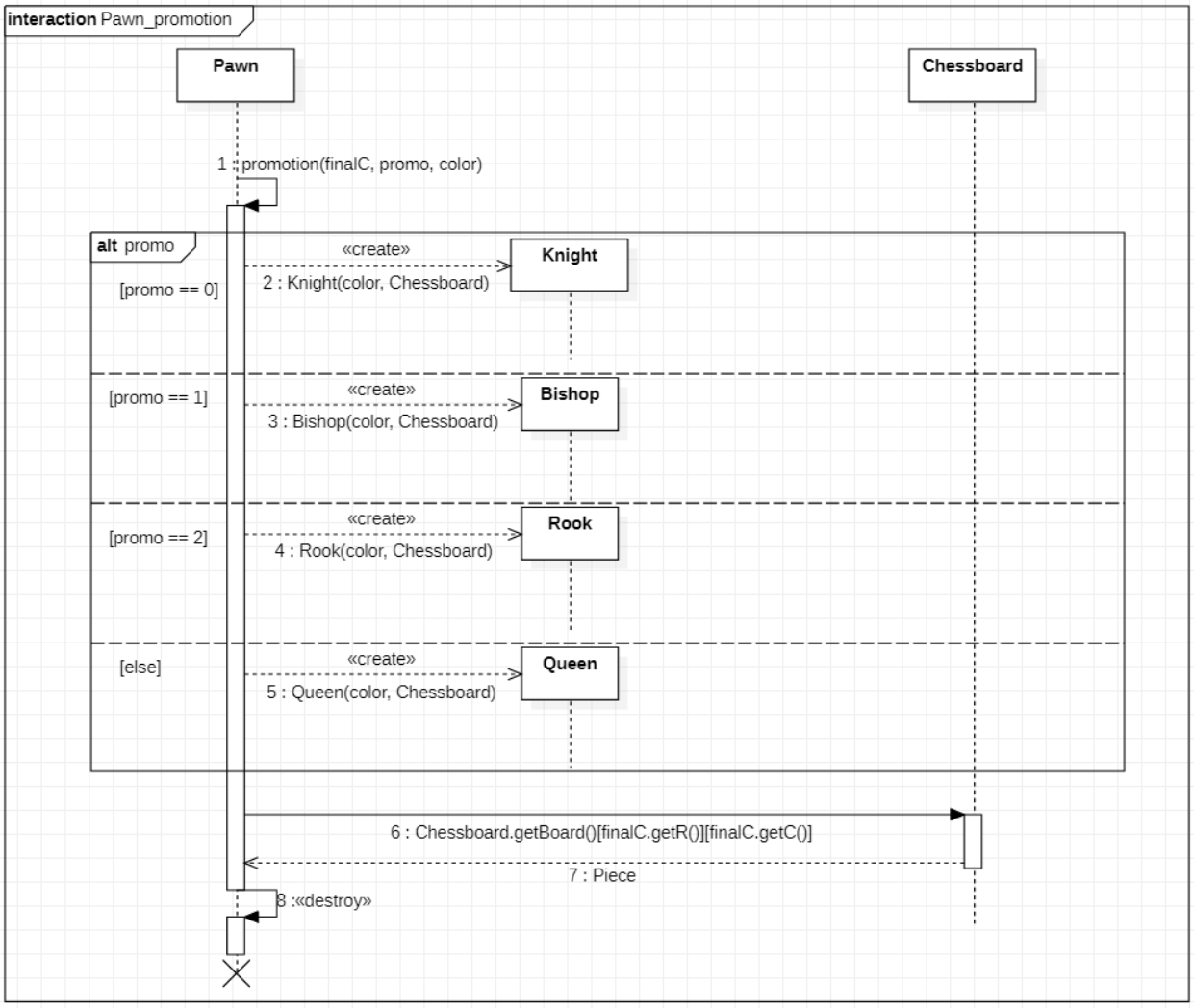
-

- <u>King :</u> Il représente le cas particulier du **`King`**, lors du roque.

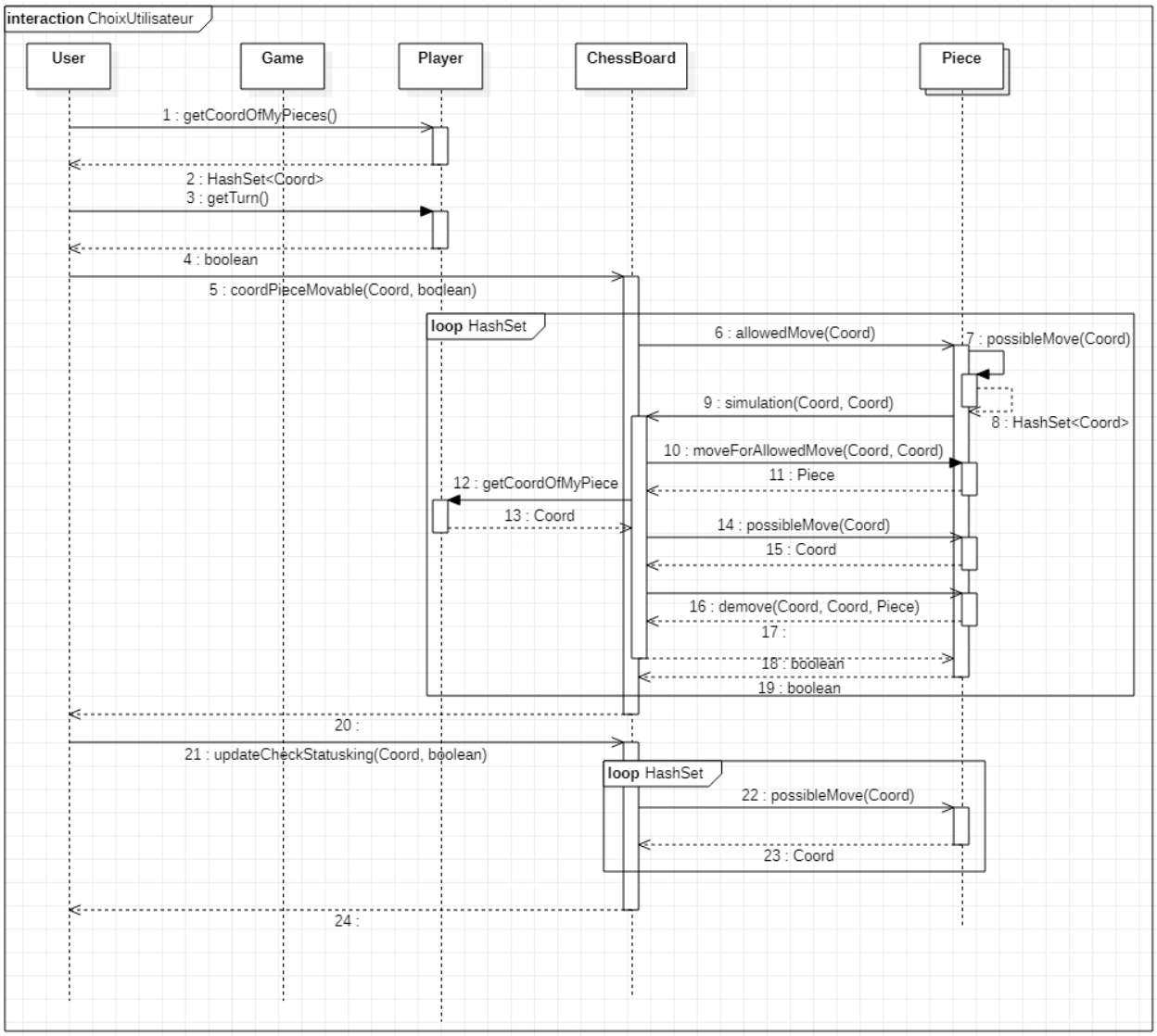


-

- Pawn Promotion : Il représente le moment où la pièce `Pawn` peut être promue en une autre pièce.



- choixUtilisateur : Il représente la génération des pièces bougeables pour le joueur actif.



3. CHOIX D'IMPLÉMENTATION DU COEUR APPLICATIVE

a) Classe Piece

La classe **`Pièce`** est la base de toutes les autres pièces. Elle transmet, via un héritage, tous ses attributs et méthodes que chaque pièce particulière pourra réimplémenter ou simplement réutiliser. Lors de la réalisation du diagramme de classe nous avions décidé de mettre la classe **`Piece`** en classe abstraite car le programme n'a jamais besoin de créer un objet **`Pièce`** directement. Chaque pièce à donc un attribut **`color`**, sous forme d'un booléen, et une référence à une instance de **`Chessboard`**, que nous détaillerons lors de la partie b). Il y a également des méthodes communes, par exemple toutes les pièces peuvent bouger, donc la méthode **`move`** est mise dans la classe **`Piece`**.

Ensuite nous avons codé, pour chaque type de pièces, une classe qui hérite de la classe **`Piece`**. Nous avons donc six classes supplémentaires. Celles-ci implémentent donc leurs déplacements particuliers, **`possibleMove`**, ainsi que l'affichage des pièces avec la méthode **`toString`**. Mais certaines pièces doivent redéfinir la méthode **`move`** car elles ont certaines spécificités.

Par exemple, le pion est la première d'entre elles. Parmi les règles particulières aux pions, nous avons la prise en passant et le déplacement de deux cases lors du premier mouvement d'un pion. Etant donné que la prise en passant consiste à autoriser un coup (la prise en passant) uniquement pendant un seul tour (on peut faire la prise uniquement au tour suivant lorsqu'un joueur avance son pion de deux cases). Il nous a donc semblé intéressant d'ajouter un attribut **`nbCoup`** (un int) dans **`Game`** et un attribut **`pawnStat`** (un int) dans **`Pawn`**. Ainsi, **`pawnStat`** va servir à la fois pour la prise en passant et l'avancé de deux cases. En effet, si **`pawnStat`** vaut 0, alors le pion n'a jamais bougé et donc peut avancer de deux cases, si l'attribut vaut **`nbCoup`**, cela signifie que le pion peut faire une prise en passant, toutes autres valeurs implique que c'est un pion "normal", qui ne peut ni prendre en passant ni avancer de deux cases.

Le pion a une dernière spécificité, la promotion. Cela se passe uniquement dans la méthode **`move`** de **`Pawn`** qui remplace le pion par une nouvelle pièce (sa promotion). En effet, une promotion est un déplacement normal du pion (il avance d'une case ou fait une prise en diagonale). Il était donc pas nécessaire d'y faire attention dans **`possibleMove`** d'autant plus que peu importe la promotion, cela n'impacte pas **`allowedMove`** (une pièce ne peut contrer l'échec de son roi que en tant que "bouclier" peu importe les déplacements de la pièce). Ces trois règles spécifiques au pion impliquent

ainsi la redéfinition de la méthode **move**.

On notera la présence du paramètre promotion, dans le **move**. Ce paramètre est essentiel pour le **Pawn**, car cela lui permet d'appeler la méthode **promoteDialog**, et ainsi pouvoir promouvoir un **Pawn**, lorsqu'il atteint l'autre bord du plateau. La méthode **promoteDialog** doit être instanciée par n'importe quelle interface (en ligne de commande ou graphique) car ces dernières doivent hériter de l'interface **ChessGameInterface**. C'est l'intermédiaire entre l'utilisateur et le programme. En effet, la promotion va casser le déroulement habituel du jeu pour inviter l'utilisateur à prendre une décision sur son **Pawn**.

L'autre pièce particulière est le roi. L'une des règles des échecs concernant le roi est bien évidemment le roque. Un roi ne peut roquer s'il n'a jamais été déplacé. Il était donc évident de mettre un attribut **castlingKing** (un boolean) dans **King** pour savoir si le roi a bougé. De même, le roi ne peut roquer si la tour concerné à déjà bougé, d'où la nécessité de l'attribut **statRook** (un booléen) dans **Rook**. Enfin, la dernière condition pour roquer est que, lors de son roque le roi ne doit passer par aucune case attaquée. Pour cela, on utilise l'attribut **coordOfMyPiece** du joueur adverse afin de voir l'ensemble des cases qu'il attaque (boucle en appelant **possibleMove** des pièces concernées (figures n°3 page 12) et on vérifie si les cases problématiques (traversé du roi) font parties des cases attaquées par l'adversaire.

De plus, le roi redéfinit aussi la méthode **move** car lors de son mouvement, elle peut impliquer le mouvement d'une des tours (en cas de roque). Elle doit aussi mettre à jour son attribut **castlingKing** en cas de mouvement. De même, la tour doit redéfinir la méthode **move** pour mettre à jour son attribut **statRook** en cas de mouvement.

```
HashSet<Coord> coordPieceBlack = this.getCb().getGame().getBlackPlayer().getCoordOfMyPieces();
for (Coord s : coordPieceBlack) {
    HashSet<Coord> tmp = new HashSet<>();
    tmp = getCb().getBoard()[s.getR()][s.getC()].possibleMove(s);
    cAttacked.addAll(tmp);
}
```

Figure n°3: Exemple de boucle pour la vérification des cases attaquées par le joueur noir.

b) Pièce bougeable et déplacements des pièces

Il nous a été demandé d'afficher les mouvements légaux d'une pièce, une fois qu'on clique sur l'une des pièces bougeables, c'est-à-dire les pièces qui peuvent bouger durant ce tour en respectant l'ensemble des règles du jeu. Pour cela, nous sommes partis du principe que dans un premier temps, toutes les pièces pouvaient se déplacer sur l'ensemble des cases de l'échiquier. Or ce n'est pas le cas dans la réalité.

Un premier tri consiste alors à regarder les déplacements possibles en fonction de la pièce en question (Exemple : uniquement les diagonales pour les fous, uniquement les cases horizontales et verticales pour la tour). On a donc décidé de matérialiser ce premier tri par la redéfinition de la méthode **`possibleMove`** pour l'ensemble des pièces (méthode abstraites de **`Piece`**).

Un second tri se fait alors à partir des résultats de ce premier tri. Ce dernier consiste à restreindre les déplacements possibles en fonction de la position de l'ensemble des pièces à un instant t. En effet, une pièce quelconque en plein milieu de l'échiquier sur le chemin de la dame l'empêche forcément d'atteindre les cases derrières la pièce gênante. On a donc choisi de matérialiser ce second tri à travers la même fonction **`possibleMove`** qui a servi au premier tri. En effet, une pièce "gênante" n'a pas le même impact en fonction des différentes pièces. Par exemple, contrairement à l'exemple précédent avec la dame, pour notre cavalier, une pièce "gênante" ne peut que impacter une seule case par rapport au premier tri. Il est donc logique de continuer à utiliser le principe de polymorphisme grâce à la relation de généralisation entre **`Piece`** et nos différentes pièces.

Enfin, le troisième tri consiste à passer des déplacements "possibles" d'une pièce à ses déplacements "légaux". En effet, au échec on ne peut jouer un coup en laissant son roi en échec. Ce troisième tri sert donc à respecter cette règle. Or, peu importe le type de pièce, un déplacement d'une pièce devient légal si son déplacement ne met pas le roi en échec, ou s'il parvient à parer l'échec en cas d'échec du roi. "Peu importe le type de pièce" nous démontre donc qu'il n'est pas pertinent que **`legalMove`** soit une classe abstraite de **`Piece`**. De plus, elle ne sera redéfini pour aucune des pièces. D'où la division de ces trois tris en deux méthodes. De plus, la méthode **`possibleMove`** sera réutilisée à d'autres moments (pour voir l'ensemble des cases attaquées par l'un des joueurs (figure n°3). Il était donc nécessaire de diviser le tri.

Ensuite, l'une des demandes du projet est d'afficher les fameuses pièces bougeables au début de chaque tour. On s'est donc dit que l'ensemble des calculs afin d'aboutir aux

coups légaux de chacune des pièces devaient se faire en préalable c'est-à-dire au début de la partie et après chaque coup. Cela se fait ainsi par l'appel de la méthode **coorPieceMoveable** qui consiste à regarder l'ensemble des pièces du joueur qui s'apprête à jouer. Afin de ne pas regarder toutes les cases du plateau pour vérifier si la case est occupée par une pièce puis la couleur de la pièce. On a donc décidé d'ajouter un attribut **coordOfMyPieces** dans la classe **Player**. De plus, cet attribut nous sera utile pour implémenter la "nul par manque de matériel". Pour revenir à notre propos, pour chaque coordonnée présent dans cet attribut, on regarde les déplacements légaux de la pièce présente sur cette coordonnée par la méthode des multiples tris vu précédemment.

Une fois ces calculs faits, et afin de s'assurer que le joueur respecte bien les règles du jeu sans pour autant tout revérifier, il fallait donc stocker ces informations dans un attribut. Ainsi, la classe **Piece** possède un attribut **allowedMove** qui sera mis à jour à chaque fois que l'on fait les calculs pour avoir les pièces bougeables (début de partie et après chaque coup). De même, pour les pièces bougeables lors d'un tour, nous avons créé un attribut **coorPieceMoveable** dans la classe **ChessBoard**.

Pour notre dernier tri afin de passer de **possibleMove** à **allowedMove** et donc prendre en compte l'échec au roi. On a décidé d'effectuer les coups de **possibleMove** et de regarder si le roi du joueur actif est en échec suite à son action. Si c'est le cas, le coup n'est pas possible et ne sera donc pas dans **possibleMove**. Sinon, le coup est possible et devient donc un coup légal. Afin d'effectuer les coups, on pouvait soit cloner l'échiquier et faire le coup mais cette solution nous semblait trop coûteuse. On a donc opté pour une simulation du coup directement sur l'échiquier du jeu. On va donc faire le coup, faire les opérations nécessaires (cases attaquées par l'adversaire et vérifier si le roi est sur l'une de ces cases (figure n°3)), on avait donc besoin de connaître la case du roi mais on ne voulait pas parcourir toutes les cases ou même **coordOfMyPieces**. Nous avons donc décidé d'ajouter une redondance de l'information en ajoutant les coordonnées des deux rois dans **ChessBoard.** Suite à cette simulation, il faudra tout remettre en place. Pour cela, on a opté pour une méthode **simulation** dans **ChessBoard** qui appelle **moveForAllowedMove** et **demove** qui seront redéfinis pour **Pawn** et **King**. En effet, tout comme **move**, **moveForAllowedMove** doit gérer la prise en passant et le roque.

En revanche, la différence entre **move** et **moveForAllowedMove** est que comme ce n'est pas un réel déplacement, **moveForAllowedMove** ne met pas jour les **pawnStat** et **castlingKing**, on ne voulait pas prendre le risque de les modifier et de les remettre avec **demove** (risque inutile d'erreur). De plus, **moveForAllowedMove** ne gère pas la promotion, en effet la promotion n'impacte pas la simulation, on peut donc simuler une promotion

par un pion (“en tant que bouclier”).

c) Classe ChessBoard/Coord

Comme le jeu d'échecs est un plateau de 8 fois 8 cases, on a rapidement décidé de représenter le plateau par une matrice de `Piece`. Ainsi la case est `null` s’il n’y a aucune pièce. Notre plateau est constitué de pièces. Néanmoins, étant donné que dans notre conception, une pièce continue d’exister après avoir été mangée (elle est transférée à la `LinkedList capturedPiece` de `Player`). Il y a donc une relation d’agrégation entre les classes `Piece` et `ChessBoard`. De plus, comme on voulait quand même avoir accès au plateau à partir des pièces, il y a donc une relation bilatérale entre ces deux classes (qui se traduit par une référence de `ChessBoard` dans `Piece` en plus de ceux des pièces dans le `board` de `ChessBoard`). Ainsi, pour accéder à l’une des cases de notre plateau, il faut y accéder à l’aide de ces coordonnées [i][j]. Or, pour stocker une coordonnée (comme on souhaite le faire dans les attributs `coorOfMyPieces` ou `coorPieceMoveable`), il faut absolument stocker un couple de deux entiers par coordonnées (l’abscisse et l’ordonnée). Ainsi, pour stocker ce couple d'entiers, nous avons décidé de créer une classe `Coord` qui possède un attribut ligne, `row`, et un attribut colonne, `column`. Cette classe `Coord` redéfinit ainsi les méthodes `equals` et `hashCode` dans le but de stocker des références de `Coord` dans nos différents `HashSet` de `Coord`. Pour ces raisons, la création de la classe `Coord` nous parut ainsi essentielle.

d) Classe Game

Pour que notre jeu d'échecs soit fonctionnel, il était primordial de créer une classe `Game` qui fait le lien entre l’ensemble de nos classes. De plus, c’est cette classe qui gère la cohérence des entrées. En effet, afin que notre jeu ne plante pas, les méthodes `select` et `play` lance l’exception `NotInHashSetException` et sont appelées dans des `try` afin de `catch` l'exception lancée. `select` permet ainsi de sélectionner la première pièce en vérifiant si elle est dans les pièces bougeables et `play` permet de changer de sélection ou de changer la pièce sélectionnée.

Ensuite, pour que notre jeu “tourne”, il faut forcément une condition d’arrêt. Une partie se finit soit lorsque l’attribut `coorPieceMoveable` de `ChessBoard` est vide c'est-à-dire, s’il n’y a plus aucun déplacement possible pour le joueur qui a le trait. Il y a donc soit échec et mat, soit pat (pour différencier les deux cas on regarde l’attribut `myKingStatus` du

`Player` qui n'a plus le trait). L'une des autres conditions d'arrêt est bien évidemment le timer, si l'un des joueurs à un timer qui tombe à zéro, il a perdu et la partie est finie. Enfin, la dernière condition d'arrêt est le nul par manque de matériel matérialisé par la méthode `checkLackStuff` de la classe `Game`.

La méthode `checkLackStuff` utilise l'attribut `coordOfMyPieces` dans `Player`, afin de regarder si les deux joueurs ne peuvent se mater l'un et l'autre. On a choisi de l'implémenter de telle sorte qu'il y a nul par manque de matériel lorsque les deux joueurs ont tous deux :

- Leur roi et un cavalier ou leur roi et un fou
- Uniquement leur roi

e) Choix de Collection

Nos collections qui contiennent des coordonnées (`instanceof Coord`), comme par exemple, `coordOfMyPieces` ou `coordPieceMoveable`, ont pour objectifs d'ajouter des coordonnées, d'en supprimer et d'en vérifier l'appartenance. De plus, il n'est pas nécessaire d'avoir de doublon d'où l'utilisation d'un `Set`. C'est la raison pour laquelle nous avons utilisé des sets et plus particulièrement des `HashSets` pour avoir l'ajout/la suppression/l'appartenance en temps logarithmiques.

En revanche, nous avons utilisé une `LinkedList` pour l'attribut `capturedPieces` de `Player`. En effet, cet attribut va constamment changer de taille. De plus, on va stocker l'ensemble des pièces capturées par le joueur et non juste qu'une instance de chaque pièce.

f) Mise à jour plateau

Une fois qu'un coup (qui est donc valide) est joué, c'est la méthode `update` de `ChessBoard` qui permet de tout mettre à jour, c'est à dire :

- Mettre la pièce sur la coordonnée d'arrivé et la retirer de la coordonnée de départ
- Mettre à jour `capturedPieces` si une pièce est capturé
- Mettre à jour `coordOfMyPieces` des joueurs
- Mettre le statut du roi du joueur qui vient de jouer à false. En effet, comme il vient de jouer un coup valide, son roi n'est forcément pas ou plus en échec.

De plus, dans le cahier des charges, il est aussi demandé d'afficher le statut des deux rois. On a donc ajouté l'attribut **myKingStatus** dans **Player** et la méthode **updateCheckStatusKing** dans **ChessBoard**. Cette méthode regarde les pièces attaquées par l'adversaire et si le roi est sur l'une de ces cases afin de mettre à jour l'attribut correspondant. L'attribut **myKingStatus** nous servira en fin de partie afin de différencier un échec et mat et un pat.

4. INTERFACE HOMME MACHINE

Avant de créer une interface graphique, nous avons implémenté une interface homme machine en ligne de commande afin de nous aider à coder. Nous sommes ensuite passé à la partie interface graphique une fois que tout le noyau applicatif était finalisé.

a) Interface console

Ainsi, pour prendre les entrées du clavier, nous avons décidé de créer une classe **Input** comportant des méthodes statiques comme par exemple **askValidCoord**. Ensuite, pour afficher les informations utiles dans l'invite de commande, il nous a fallu redéfinir la méthode **toString** de nos différentes classes. De plus, afin d'augmenter l'ergonomie de notre application (même pour l'interface graphique), nous avons créé un attribut static **configBoard** dans **ChessBoard**. Ce dernier nous permet ainsi de passer des coordonnées de la matrice [i][j] aux "vrai" coordonnées d'un jeu d'échec (comme par exemple la case 'e4') aussi bien dans le **toString** de **ChessBoard** que dans les entrées de **Input**.

b) Interface graphique

Nous avons codé notre interface graphique selon le modèle PAC (Présentation, Abstraction, Contrôle). La facette abstraite (Abstraction) est notre noyau applicatif (package **global** et **pieces**). La facette présentation est notre package **interfaces.** Enfin, notre package **controlers** fait le lien entre ces deux facettes. Ainsi chaque modification est gérée par un contrôleur et elle est propagé au sein des contrôleurs. Ensuite, pour "dérouler" notre partie d'échec, il a fallu créer une méthode

courseOfTheGame dans **Game** qui a pour fonction d'appeler l'ensemble des méthodes nécessaires pour qu'un joueur joue un coup, mettre à jour le jeu et de faire cela tant que la partie n'est pas finie.

Figure n°4: Interface finale

L'organisation de notre scène se fait comme suit : en résumé, nous avons un **BorderPane** qui contient en son centre le plateau (un **GridPane**), en haut nous avons le "bandeau" du joueur noir et en bas celui du joueur blanc. Enfin, à droite de notre **BorderPane** principal, nous avons l'ensemble des fonctionnalités supplémentaires qu'offre notre application.

Dans la classe **Graphic**, les méthodes principales sont **loadGame, newGame, displayCell, numberingRowCol**. La première contient tous nos outils JavaFx qui permettent d'afficher notre interface, elle sera appelée plusieurs fois pour charger la partie. La seconde est une extension de la première méthode, elle sert à réinitialiser une nouvelle partie (bouton new). Nous avons séparé ces deux méthodes pour les différentes fonctionnalités dans le jeu. La troisième méthode sert à afficher les cases dans l'échiquier et la dernière est pour

afficher le nom des colonnes (A-H) et le nom des lignes (1-8).

Ensuite, pour les “mises à jours”, nous avons décidé de créer un contrôleur appelé **ControleTileStack** qui lie un enfant de notre **GridPane grid** (le tablier) à une case du **board** de **ChessBoard** (il prend dans une coordonnée **Coord**). C’est notre contrôleur principal car c’est lui qui va gérer les entrées (cliques de la souris) en implémentant **EventHandler<MouseEvent>**. Ainsi, à chaque clique sur l’un des enfants de **grid**, la méthode **handle** remplace la méthode **courseOfTheGame** de l’interface en ligne de commande. C’est donc pour cela que ce contrôleur doit posséder une référence vers **Game** et vers **Graphic** afin d’avoir toutes les informations nécessaires pour gérer le bon fonctionnement de notre jeu.

Alors, pour créer cette interface, nous allons découper la fenêtre comme dans la figure n°4, en 4 parties avec l’aide d’un **BorderPane screen**.

La partie du haut et du bas sont respectivement destinées pour le joueur noir et blanc et qui sont tous les deux créées par la méthode **playerBorder**. Cette méthode affiche 3 informations verticalement avec l’aide d’un conteneur vertical (**VBox**). L’information du haut affiche le nom des joueurs, le statut des rois des joueurs, le temps écoulé du jeu et ils sont à leur tour placés sous forme des conteneurs horizontaux (**HBox**). L’information au milieu affichera le nombre des pièces perdues. Nous avons choisi de séparer l’information du haut et l’information du bas pour éviter un conteneur trop long en haut et en bas de BorderPane **screen** et déformer l’affichage de ce dernier. Finalement, l’information du bas affiche les pièces perdues par le même joueur sous forme d’un conteneur de flux (**FlowPane**) pour pouvoir afficher la pièce l’une après l’autre. Pour pouvoir afficher ces informations, nous avons décidé de créer les contrôleurs dans le package **controlers** qui puissent créer un lien entre les méthodes dans les classes existantes dans **global** et les méthodes dans **interfaces.** Par exemple, **ControleKingStatus** sert à vérifier si le roi est en échec, ce qui est notifié par la classe **Player** comme **Observer**. Si ce statut du roi est changé, elle mettra à jour le **TextField** qui correspond.

La partie à droite est destinée pour les boutons placés l’un en dessous de l’autre verticalement, elle est donc créée par la méthode **createSaveOpenExitButton**. D’abord, nous avons le bouton “New” qui appelle la méthode **newGame** pour créer une nouvelle partie. Le bouton “Save” est pour la sauvegarde d’une partie. Le bouton “Open” est pour charger une partie avec l’aide de la méthode **loadGame** . Le bouton “Exit” est pour quitter

la plateforme du jeu. Le bouton “Turn” est pour tourner le plateau, elle appelle donc les méthodes **`displayCell`** et **`numberingRowCol`**. Et enfin, l’affichage des tours passés et l’affichage de l’horloge, qui permettent respectivement à l’utilisateur de visualiser le nombre de tours passés et le temps total passé.

La partie au milieu est destinée pour l’affichage de l’échiquier. Nous l’avons découpé encore une fois en 3 sous parties grâce à un **`BorderPane`**. La sous partie à droite est initialement l’affichage des rangs de 1 à 8 de l’échiquier. La sous partie en bas est initialement l’affichage des colonnes de A à H de l’échiquier. La sous partie au milieu est finalement pour l’échiquier. Nous avons donc choisi de créer cet échiquier sous forme d’un **`GridPane`** qui stocke les 64 cases grâce à la méthode **`displayCell`** qui génère les cases de type **`StackPane`**. **`StackPane`** est utile pour empiler les différentes couleurs ainsi que l’image des pièces qui correspondent pour chaque cases.

5. AXES D'AMÉLIORATION

a) Repenser la conception

i) Les relations

Nous avons beaucoup hésité sur la manière de savoir si un coup possible était un coup légal (si après le coup son propre roi était en situation d’échec, si oui le coup n’est pas légal). De fait, avant d’opter pour la solution d’une simulation pour vérifier cela, nous avions tout d’abord penser à une manière originale de répondre à ce problème en définissant des “relations”. Cette implémentation répondait à un enjeu simple : elle doit être la moins coûteuse possible.

L’idée est simple : maintenir des liens entre un **`King`**, et n’importe quelle autre pièce. Nous pourrions ainsi pondérer ce lien (via un attribut **`degre`**), qui représenterait la proximité entre ces deux pièces, c'est-à-dire le nombre d'obstacles. Ce degré pourrait prendre des valeurs entre 0 et 6. Par exemple, une relation de degré 0 définit un échec, car il y a 0 pièce entre un **`King`** et la **`Piece`** mise en relation. On définirait alors une relation comme un objet stockant une **`Piece`**, un entier (que l’on appellera “degré”) ainsi qu’une collection de **`Coord`**. Cette dernière stockera toutes les positions entre les deux pièces, pour ainsi faciliter les calculs.

Ainsi, à chaque instant, l'application pourrait observer toutes les relations en place pour voir si certains déplacements sont possibles, rien qu'avec le parcours d'une liste réduite. Elle prendrait par exemple en compte l'appartenance d'une **`Piece`** à une autre relation d'une autre **`Piece`** adverse ayant un degré de 1 afin d'éviter un mauvais déplacement.

De même, lors de déplacement de **`Piece`**, l'application aurait la possibilité entre : créer une relation, la conserver, la mettre à jour, ou la supprimer. Seul le cas d'un déplacement du roi peut s'avérer plus coûteux, car le système devrait casser toutes les relations existantes avec ce roi, afin d'en recréer de nouvelles. Elle ferait ainsi le calcul de tous les déplacements possibles (**`possibleMove`**) de l'adversaire.

En conclusion, les calculs sont réduits, car toutes les informations nécessaires se trouvent dans des listes de relations, qui suffira de lire. Cependant le principal défaut de cette implémentation est en réalité sa complexité de mise en place, et sa maintenance. Car les relations doivent être générées au fur et à mesure de la partie, ce qui empêche une mise en place de plateau originale, notamment.

ii) La mise en place des timers

Nous pourrons également nous attarder sur la mise en place du temps au sein de l'application, qui pourrait être repenser. En effet, la majorité de notre application respecte les normes PAC. Néanmoins, les horloges des joueurs ainsi que de la partie ont été implémentés vers la fin du projet, ce qui n'a pas permis une mise en place aussi flexible. Il y a en effet un **`ControleTime`**, qui est relié à l'affichage. Cependant, il manque la partie abstraction, qui a été partiellement faite. Nous retrouvons effectivement un objet **`SetTimer`**, qui pourra représenter l'abstraction (les timer ne sont pas implémentés dans le noyau applicatif). Cette abstraction est donc en quelque sorte détachée du reste, et ne permet pas une aussi grande flexibilité, en cas d'ajout ou de modification de fonctionnalités.

b) Forces de notre code

Nous avons finalement réussi à concevoir un jeu d'échec fonctionnel qui implémente l'ensemble des demandes du cahiers des charges :

- Afficher les pièces bougeables à tout instant
- Afficher les déplacements de la pièce lorsqu'on clique sur une pièce bougeable

- Chaque joueur possède un timer afficher à chaque instant
- Chaque joueur peut voir ses pièces perdues à chaque instant
- Chaque joueur peut voir le statut de son roi à chaque instant
- Les joueurs peuvent directement relancer une nouvelle partie (bouton new)
- Les joueurs peuvent sauvegarder une partie et la recharger ultérieurement
- Le plateau peut tourner de 90° selon l'envie des joueurs
- Le nombre de pièce restant de chaque joueur
- Afficher le temps cumulés des deux joueurs
- A la fin de la partie, le résultat est affiché

De plus, grâce à l'utilisation des Exceptions, cela limite au maximum les problèmes tels que les bugs/plantage de notre jeu d'échecs.

Ce qui fait la force de notre code est sa flexibilité, d'une part par son noyau applicatif, mais également via son intégration avec le modèle PAC. De plus, nous avons fait attention à produire une structure cohérente, qui évite le maximum de répétition. Le code est donc à la fois lisible et facile à maintenir.

c) Améliorer les fonctionnalités de notre jeu

Néanmoins, notre projet est loin d'être parfait dans la mesure où bon nombre de fonctionnalités peuvent encore être implémentées. Par exemple :

- Implémenter la règle du nulle par répétition (50 coups d'affilés sans mouvement de pion ou prise, ou se retrouver trois fois dans exactement le même pattern)
- Laisser les joueur choisir le temps du timer au début d'une partie
- Faire des timers avec gain de temps lorsqu'un coup est joué
- Interface graphique encore plus ergonomique et réutilisable (meilleure mise en oeuvre du modèle PAC)
- Faire une interface plus jolie et plus personnalisé (On est allé au plus simple par manque de temps)

6. CONCLUSION

Pour conclure, ce projet en génie logiciel permet de voir l'évolution d'un projet de la conception UML à l'interface graphique en passant par le noyau fonctionnel, chose que nous ne pouvons faire en TD. De plus, nous avons essayé de rendre notre projet ergonomique et réutilisable.

Finalement, nous pensons avoir respecté la quasi-totalité voir la totalité du cahier des charges du projet. Nous sommes ainsi assez fiers de notre rendu dans la mesure où nous avons réussi à implémenter une partie d'échec jouable avec l'ensemble des règles du jeu d'échec (même les plus complexes) et pas mal d'autres fonctionnalités. En revanche, nous aurions aimé rendre une version finale plus qualitative avec des fonctionnalités supplémentaires qui font la différence (cf partie Axes d'amélioration, améliorer les fonctionnalités de notre jeu).