1 Introduction

Le but de ce laboratoire est d'implémenter un jeu d'échec fonctionnel. Cela implique de gérer les déplacements des pièces, les règles du jeu, la prise des pièces, le déplacement de pièce tour par tout, les échecs et mat, les roques, la promotion, la prise en passant, etc. Une interface graphique a été mise à notre disposition pour pouvoir afficher le jeu correctement. Nous avons donc implémenté les fonctionnalités demandées, les différentes classes pour modéliser des pièces, les différents mouvement possible et nous avons essayé de gérer les erreurs qui peuvent survenir.

2 Diagrammes de classes

2.1 Package `engine`

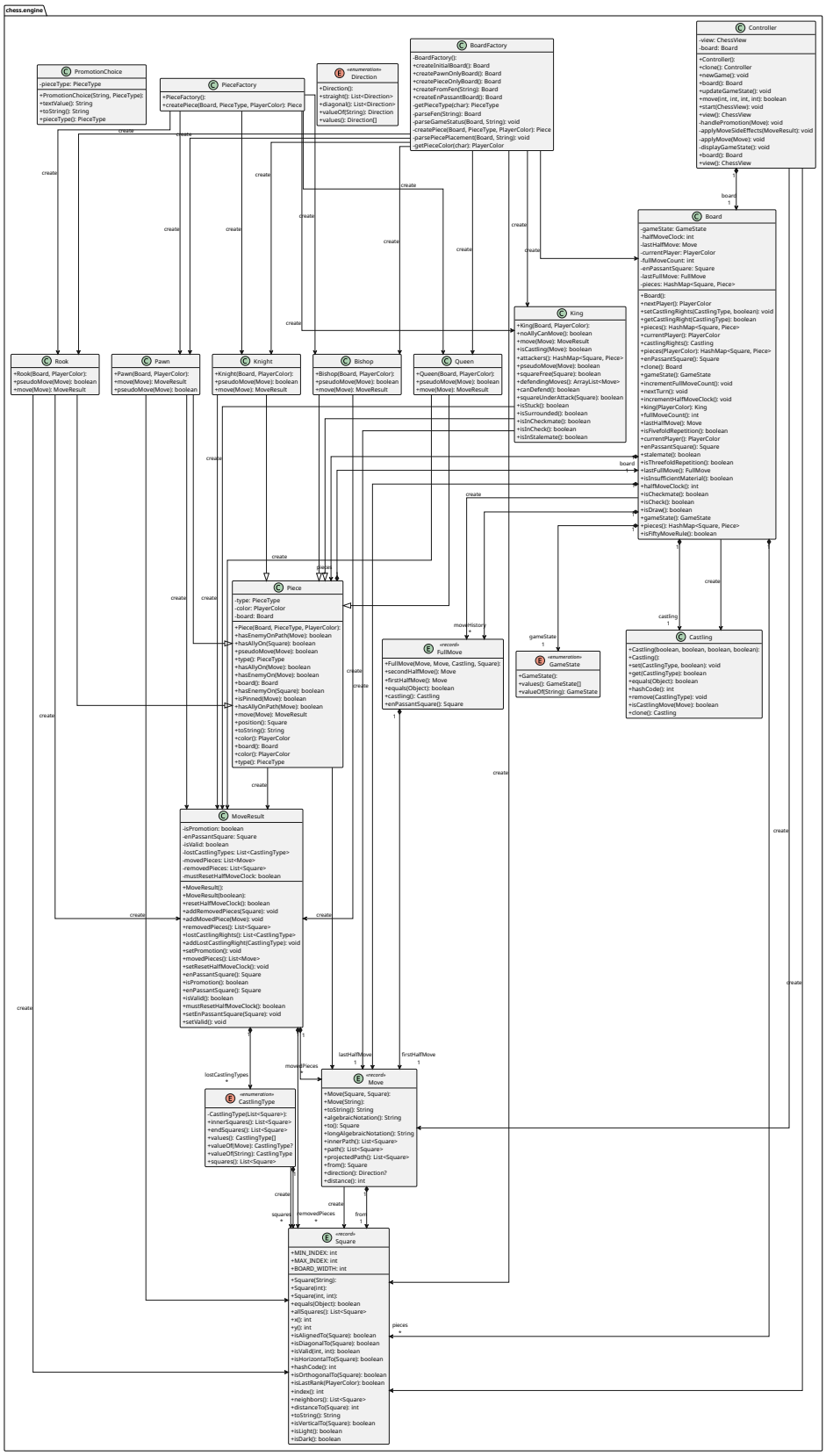

FIGURE 1 – Modélisation d'un moteur d'échecs.

2.2 Package `board`

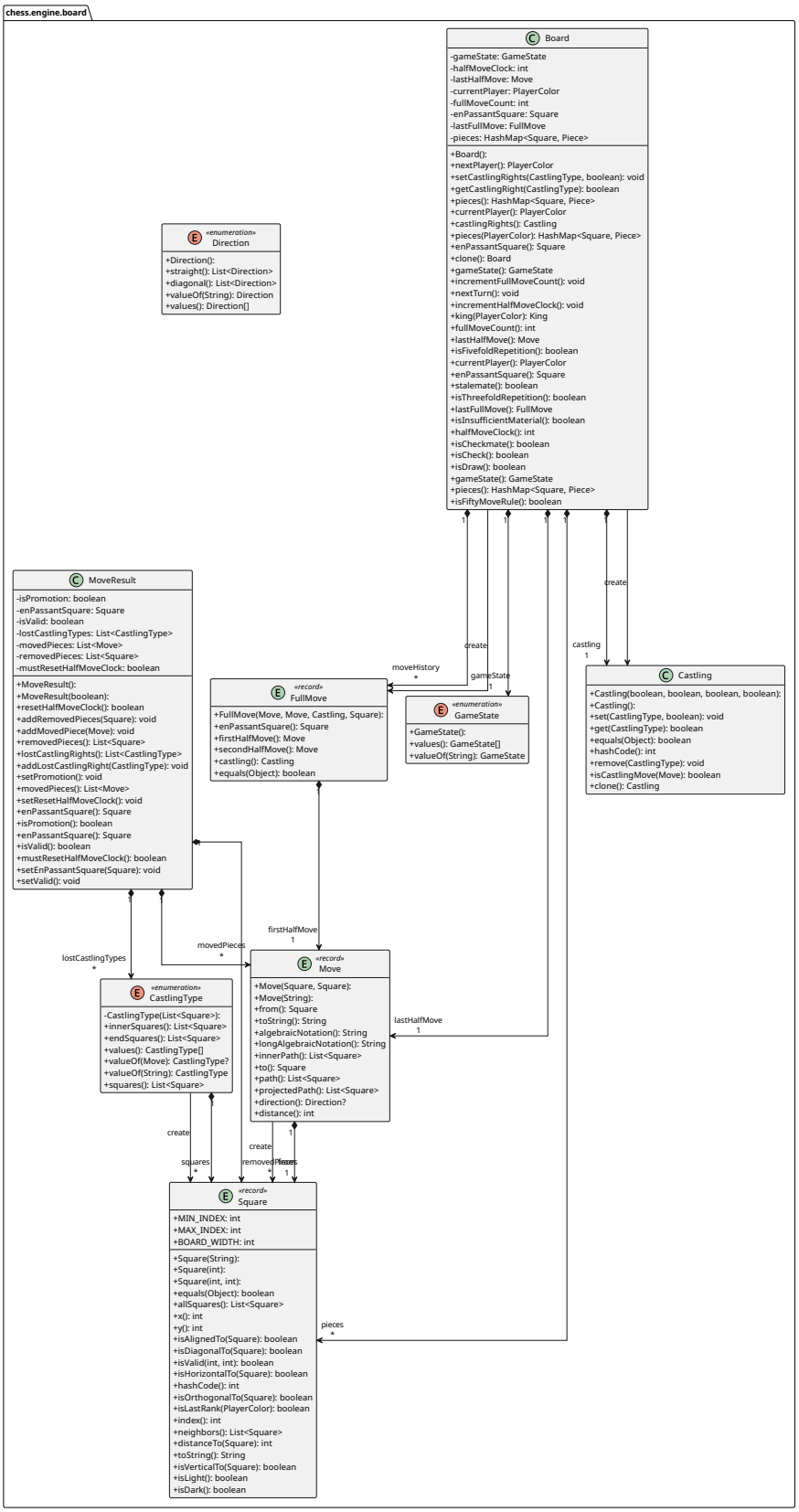

FIGURE 2 – Modélisation des modèles qui encapsulent les données du jeu d'échecss.

2.3 Package `piece`

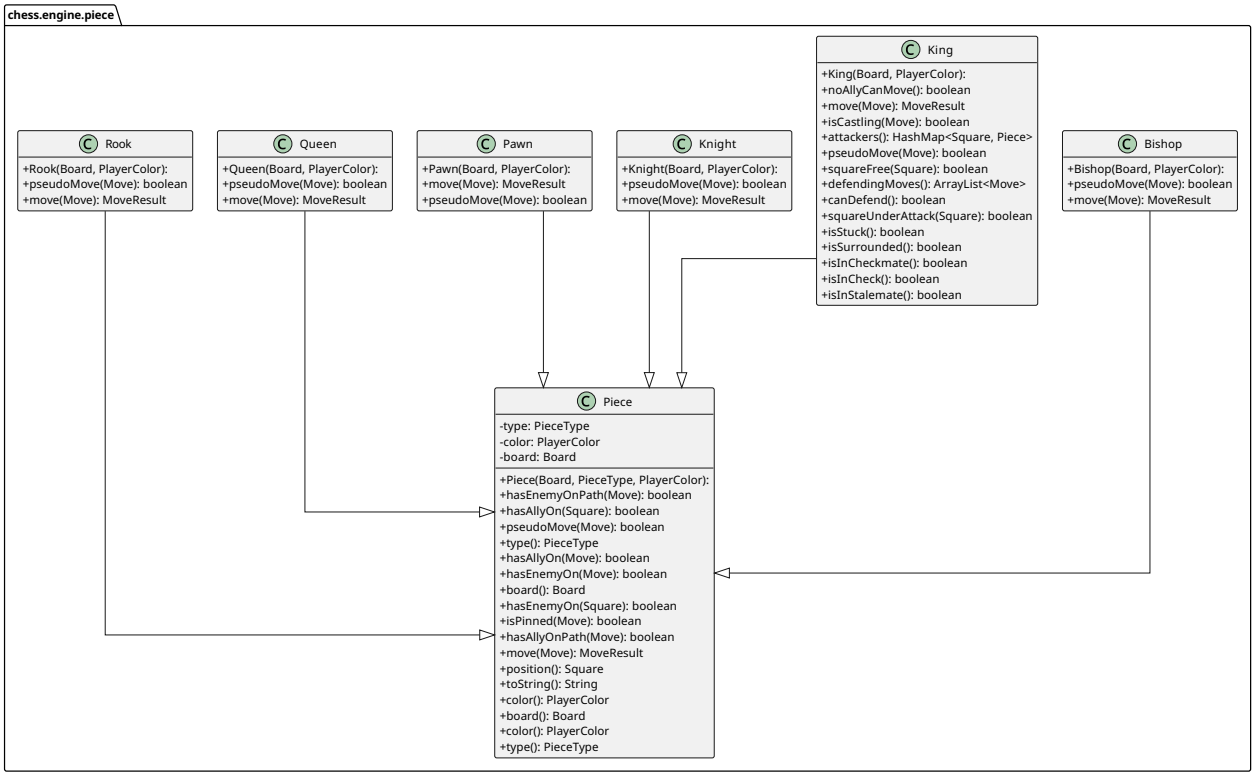

FIGURE 3 – Modélisation des pièces du jeu d'échecs.

2.4 Package `util`

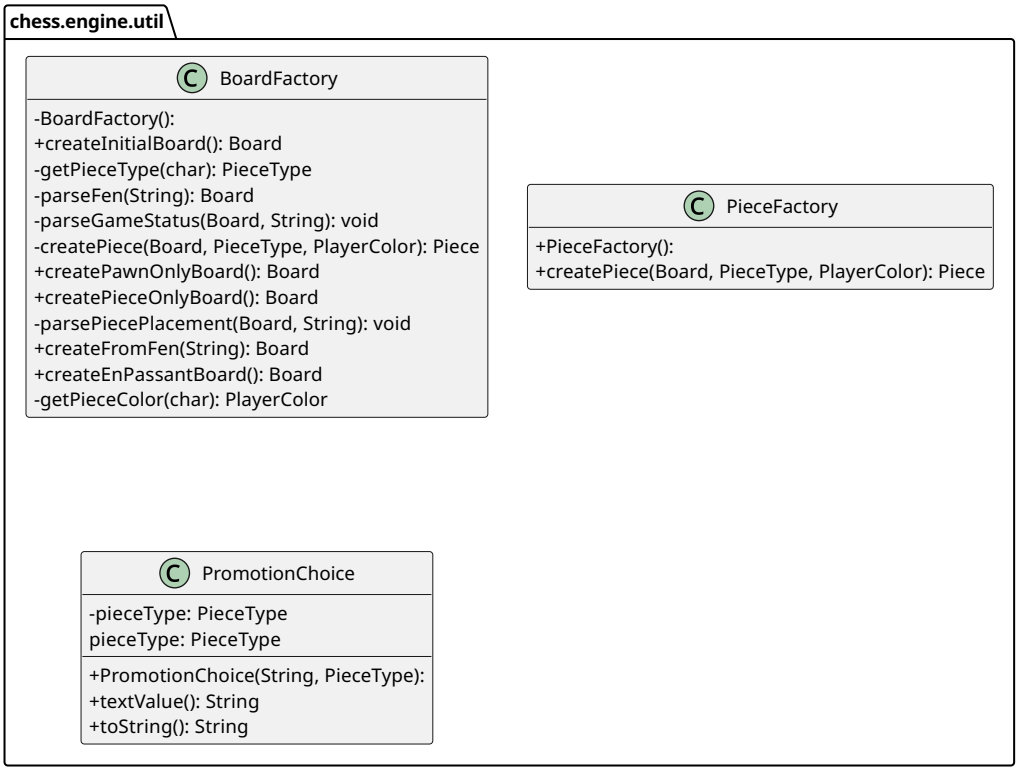

FIGURE 4 – Modélisation des classes utilitaires du moteur d'échecs.

3 Implémentation

Le laboratoire a été réalisé avec les hypothèses de travail et choix d'implémentation détaillés dans les sections suivantes.

3.1 Modèle

3.1.1 Classe Board

La classe Board permet de modéliser le plateau de jeu. La classe utilise une HashMap<Square, Piece> pour stocker les pièces d'échecs en fonction de leur position sur le plateau. Chaque clé de la HashMap est un objet Square représentant une position sur le plateau, et chaque valeur est un objet Piece représentant une pièce d'échecs. La classe maintient une liste moveHistory, qui enregistre l'historique complet des mouvements effectués pendant la partie et est utile par exemple pour détecter les répétitions de mouvement. Elle gère également les compteurs de demi-coups et de coups complets pour suivre la progression du jeu. Elle va aussi gérer le tout par tour des joueurs en gardant une trace du joueur actuel (blanc ou noir) grâce à la méthode nextTurn() qui gère le passage au tour suivant, en changeant le joueur actuel et en incrémentant le compteur de coups complets. La classe gère s'occupe aussi des droits de roque pour chaque joueur avec castlingRights et suit la case sur laquelle une capture en passant est possible avec enPassantSquare. Finalement, elle vérifie divers états du jeu, tels que l'échec, l'échec et mat, la nulle par répétition de position ou par la règle des cinquante coups. Cette classe agit comme un modèle central pour la gestion des états d'un jeu d'échecs, en gérant les positions des pièces, les mouvements, les tours de jeu, et les règles spéciales.

3.1.2 Classe Square

Square est un record qui permet de modéliser une case du jeu d'échec. La logique du constructeur vérifie que la case soit légal et donc dans le board. Elle contient aussi des méthodes pour savoir si un autre square est giagonale, vertical ou horizontale ainsi que la distance d'un square à l'autre. On peut aussi récupérer toutes les cases autour.

3.1.3 Classe Move

Move est un record qui permet de modéliser un movement donc d'une case from à une case to. Plusieurs méthodes sont fournies afin d'avoir une écriture algébrique ou algébrique longue ainsi que des méthodes pour récupérer une liste de Square qui représente le chemin entre deux cases (‘path‘, ‘innerPath‘ et ‘projectedPath‘).

3.1.4 Classe FulleMove

Un full move est un record qui permet de modéliser un mouvement complet. Utilisé pour détecter un cas de répétition de mouvement. Elle prends aussi en paramètre un catlingRights et une case qui représente le movement enPassant.

3.1.5 Classe GameState

Simple enum pour représenter les différents états du jeu.

3.1.6 Classe MoveResult

La classe MoveResult est un container qui permet de stocker les effets secondaires d'un mouvement. Elle contient un booléen qui indique si le mouvement est légal, la nouvelle case en passant si elle existe, un booléen qui indique si le mouvement est une promotion, les droits de roque après le mouvement, les pièces déplacées autres que la pièce concernée par le mouvement, les pièces capturées et la nécessité de remettre à zéro le compteur de demi-coups.

3.1.7 Classe Direction

Enum pour représenter différentes directions.

3.1.8 Classe Castling

Castling est une classe utilitaire conçue pour gérer les droits de roque dans un jeu d'échecs, offrant des méthodes pour définir, vérifier, et modifier ces droits.

3.1.9 Classe CastlingType

Enum pour distinguer les différents types de roque avec plusieurs méthodes de contrôle.

3.2 Contrôleur

La classe 'Controller' s'occupe de l'implémentation du contrôleur dans l'architecture MVC de notre jeu d'échecs. Elle va s'occuper de la gestion des parties avec les méthodes start et newGame qui s'occupent de créer un nouveau jeu avec les positions voulues. La méthode move gère le déplacement des pièces sur le plateau. Elle vérifie la validité des mouvements, gère les effets secondaires des mouvements (comme la promotion et la prise en passant), et met à jour la vue et le modèle. HandlePromotion gère la promotion des pions et applyMove et applyMoveSideEffects appliquent le mouvement principal et ses effets secondaires sur le plateau et la vue.

Il est a noter que la gestion des effets secondaires produits par un mouvement aurait pu être implémentée en utilisant un système de callback, mais nous avons préféré utiliser une approche plus simple et plus directe en faisant retourner aux méthodes `move` des pièces un objet `MoveResult` qui contient les effets secondaires à appliquer.

3.3 Classe Pièce

Pour implémenter de la manière la plus orientée objet possible, nous avons fait le choix de créer classe pièce qui est une classe abstraite et qui permet de modéliser les différentes pièces du jeu d'échec. Elle contient un board son type de pièce ainsi que la couleur ce qui permet d'identifier les différentes pièces. Elle contient aussi plusieurs setter pour ses différentes attributs. Ensuite, différentes méthodes plus avancées sont fournies :

- `move` : Cette méthode permet de déplacer une pièce sur le board.La méthode move de pièce va vérifier si le mouvement est légal et chaque pièce réimplémente cette méthode afin de bouger en fonction de leur propre règle.
- `pseudoMove` : Cette méthode permet de simuler un mouvement. Elle prend en paramètre un attribut Move.
- `hasEnemyOn` : Cette méthode permet de vérifier si une pièce ennemie est sur une case donnée.
- `hasAllyOn` : Cette méthode permet de vérifier si une pièce alliée est sur une case donnée.
- `hasEnemyOnPath` : Cette méthode permet de vérifier si une pièce ennemie est sur le chemin entre deux cases.
- `hasAllyOnPath` : Cette méthode permet de vérifier si une pièce alliée est sur le chemin entre deux cases.
- `isPinned` : Cette méthode va vérifier si la pièce que l'on souhaite bouger est épinglée à sa place, cet à dire que si elle bouge, le roi sera en échec et on ne peut pas permettre cela.

Ces différentes méthodes sont donc utilisées par toutes les pièces afin de vérifier différentes condition pour pouvoir avancer.

3.3.1 Classe Pawn

Cette classe permet de modéliser un pion. Elle hérite de la classe pièce dont la plus part des méthodes sont déjà implémentées. La plus grosse différence est qu'elle re-définit la méthode pseudo-move qui est utilisé pour faire plusieurs vérifications de mouvement et s'assurer que le déplacement souhaité est possible. Elle va donc checker si elle peut avancer s'une case, de deux cases, si elle peut capturer une pièce ou encore le en passant.

3.3.2 Classe King

Cette classe permet de modéliser un roi. En plus des méthodes move et pseudoMove qu'elle redéfinit, elle contient quelques méthodes supplémentaires qui permettent de vérifier plusieurs règles du jeu qui lui vont lui permettre de bouger tel que les fonctions de vérification d'échec, d'échec et mat, de roque, s'il peut bouger sur une case ou si elle est attaquée par une pièce ennemie.

3.3.3 Autres pièces

Les autres pièces sont modélisées de la même manière que le pion. Elles héritent de la classe pièce et redéfinissent soit la méthode Move ou soit la méthode pseudoMove en fonctions de leurs règles de déplacement.

3.4 Remarques

4 Tests

L'ensemble du jeu et des règles ont été testé manuellement et automatiquement.

4.1 Tests automatisés

Des tests unitaires réalisés avec `JUnit` valident le fonctionnement des classes implémentées. Toute une liste de tests automatisés ont été réalisé pour valider notre implémentation et aider à trouver les erreurs. Cette automatisation aide à vérifier que les méthodes des classes fonctionnent correctement, que les résultats sont ceux attendus ainsi que les différentes règles du jeu d'échec soient respectées. L'utilisation d'une batterie de tests automatique permet d'éviter les régressions et de s'assurer que les modifications apportées au code ne cassent pas le fonctionnement des classes.

Lors de l'échec d'un test, un script python est exécuté pour générer les mouvements légaux attendus trouvés par un moteur complet (ici `pychess`). Cette technique a mené à un gain de temps considérable lors du développement. Pour que le script s'exécute, il faut qu'un test échoue. Il est possible de par exemple retirer une ligne de code de l'une des classes de pièce pour que le test échoue et que le script s'exécute.

Les positions utilisées lors des tests automatisés sont inscrites dans le fichier `src/test/resources/positions.txt`.

4.2 Tests manuels

Les tests manuels suivants ont été réalisés pour valider le fonctionnement du jeu d'échecs.

Description du Test	**OK/NOK**
Le pion peut avancer d'une case ou de 2 cases s'il n'a pas encore bougé	OK
Toutes les pièces peuvent avancer selon leurs règles	OK
Lorsqu'un joueur bouge une pièce, le tour passe à l'autre couleur	OK
Les pièces peuvent capturer les pièces adverses	OK
Les pièces ne peuvent pas capturer les pièces alliées	OK
Les rois ne peuvent pas être capturés	OK
Les rois ne peuvent pas se mettre en échec	OK
Les pièces ne peuvent pas mettre leur propre roi en échec en sortant d'un épinglage	OK
Les grand et petit roques fonctionnent	OK
La promotion fonctionne	OK
La prise en passant fonctionne	OK
L'échec et mat fonctionne	OK
La partie nulle pour matériel insuffisant fonctionne	OK
La partie nulle pour pat fonctionne	OK
La partie nulle pour triple répétition fonctionne	OK
La partie nulle pour quintuple répétition fonctionne	OK
La partie nulle pour quintuple répétition fonctionne	OK
La partie nulle pour la règle des 50 coups fonctionne	OK

Table 1 – Tableau des tests manuels réalisés.

5 Conclusion

En conclusion, ce laboratoire nous a permis de mettre en pratique les connaissances acquises en cours de POO et d'implémenter une solution orientée objet à un problème concret. La séparation des responsabilités entre les différentes classes permet de modéliser le jeu d'échecs de manière efficace et de pouvoir implémenter les règles du jeu de manière modulaire.