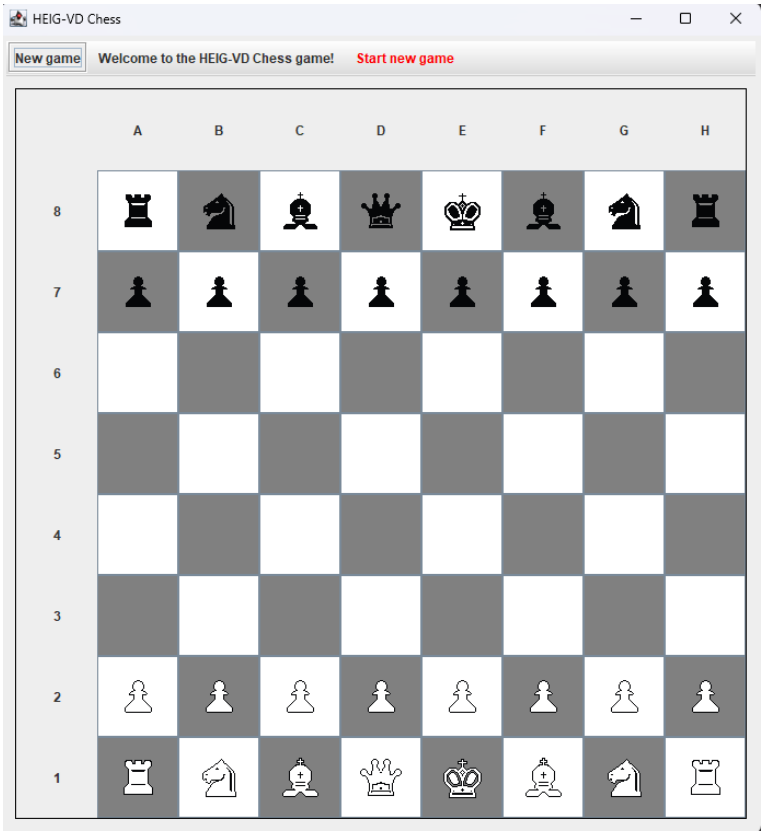


Laboratoire 8

Echec

Auteurs :
Cosmo de Oliveira Maria Vitória
Koestli Camille
Roland Samuel

Professeur :
Donini Pier
Assistant :
Decorvet Grégoire

Classe : A
Date : 10.01.2024

1 Introduction

L'objectif de ce laboratoire est de créer un jeu d'échecs, en respectant certaines règles de bases et sur un interface de jeu. Le jeu est conçu pour être joué en mode console ou avec une interface graphique utilisateur (GUI).

2 Exécution

Nous avons migré le projet vers Gradle afin de pouvoir écrire des tests automatisés avec Junit et les lancer en mode continu. Pour lancer le projet, il est possible d'utiliser Gradle.
Pour lancer le mode graphique `gradle run`
Pour lancer le mode console: `gradle run --args="-c" --console plain -q`

Si vous préférez ne pas installer Gradle, il y a également le wrapper disponible à la racine du dossier. Il suffit de remplacer dans les commandes ci-dessus `gradle` par `./gradlew` Linux/Mac ou `.\gradlew` sous Windows.

3 Explication du Code

3.1 Structure Principale

Nous avons construit notre jeu autour de plusieurs classes clés :

- **Chess**: La classe principale qui lance le jeu. Elle crée le contrôleur (`ChessController`) et la vue (`ChessView`), soit en mode console (`ConsoleView`) soit en mode GUI (`GUIView`).
- **Board**: Classe qui représente le plateau de jeu. Il gère les pièces sur le plateau, leurs mouvements, et vérifie si un roi est en échec. Nous avons aussi choisir de traiter si le roi est en échec dans cette classe. Le jeu vérifie après chaque mouvement si un roi est en échec, ce qui est crucial pour respecter les règles des échecs. En plus de la liste des pièces dans un tableau 2 dimensions, nous stockons le dernier mouvement et la dernière pièce déplacée afin de valider certaines contraintes de la prise en passant.
- **Piece**: Il s'agit d'une classe abstraite représentant les différentes pièces d'échecs. Chaque type de pièce est une sous-classe de `Piece`. Cette classe gère les mouvements valides, les collisions, et si une pièce a déjà été déplacée. La pièce connait sa position sur le plateau, cette position est tenue à jour via la méthode `Piece.setPoint()`, ainsi que son dernier mouvement défini via `Piece.setLastMove()` permettant de savoir si elle a bougé.
- **Move**: Cette classe représente un mouvement sur le plateau par un vecteur (x,y) et un maximum de fois que l'opération peut être faite, ce qui permet de décrire n'importe quel mouvement. Il contient la logique pour vérifier si un mouvement est valide et pour appliquer les changements sur le plateau. Pour les mouvements spéciaux, nous avons créé différentes sous-classes de `Move` afin d'y implémenter leurs particularités.

4 Tests réalisés

Nous avons souhaité réaliser une série de tests qui permettent de vérifier que chaque élément du jeu fonctionne comme prévu. Les tests touchent différents éléments :

- **Tests des mouvements**: Ces tests vérifient la validité des mouvements des pièces sur le plateau. Ils contrôlent que chaque déplacement est conforme aux règles.
- **Tests des conditions d'échec**: Ces tests vont contrôler si le jeu détecte correctement les situations d'échec, c'est-à-dire quand le roi est menacé et empêche tout autre mouvement.
- **Tests du roque**: Ces tests permettent de contrôler que le grand et petit roque ne se produit que lorsque toutes les conditions sont remplies (ni le roi ni la tour n'ont bougé, les cases entre eux sont libres, etc.).
- **Tests de la prise en passant**: Ces tests vérifient que ce mouvement se réalise lorsque le pion adverse a avancé de 2 cases depuis sa position de départ et se trouve à côté du pion attaquant.
- **Tests de la collision**: Ces tests vérifient que les pièves ne peuvent pas se déplacer à travers d'autres pièces, sauf pour le cavalier.

4.1 Tests console

Méthode: Exécution en mode console avec la commande suivante : `gradle run --args="-c" --console plain -q`

```
Start new game
8 |R N B Q K B N R
7 |P P P P P P P P
6 |
5 |
4 |
3 |
2 |P P P P P P P P
1 |R N B Q K B N R
-------------------
   A B C D E F G H
Next move?
e2e3
e2e3
8 |R N B Q K B N R
7 |P P P P P P P P
6 |
5 |
4 |
3 |        P
2 |P P P P   P P P
1 |R N B Q K B N R
-------------------
   A B C D E F G H

Next move?
a2a4
a2a4
```

```
C'est au tour de l'autre joueur...
Invalid move
8 |R N B Q K B N R
7 |P P P P P P P P
6 |
5 |
4 |
3 |        P
2 |P P P P   P P P
1 |R N B Q K B N R
-------------------
   A B C D E F G H
Next move?
a8a6
a8a6
Invalid move
8 |R N B Q K B N R
7 |P P P P P P P P
6 |
5 |
4 |
3 |        P
2 |P P P P   P P P
1 |R N B Q K B N R
-------------------
   A B C D E F G H
Next move?
a7a5
a7a5
8 |R N B Q K B N R
7 |  P P P P P P P
6 |
5 |P
4 |
3 |        P
2 |P P P P   P P P
1 |R N B Q K B N R
-------------------
   A B C D E F G H
Next move?
```

Note: nous n'imprimons pas les tests, car cela n'était pas utile et par économie de papiers. Il est bien sûr possible de voir le détail dans le code des tests si besoin dans le rendu Cyberlearn.

4.2 Tests sur les Mouvements

Table 1: Tests Move()

Pièce	**Peut aller vers**	**Et pas vers**
Move de Pion Blanc (1, 1)	(1, 2)	(1, 4), (0, 3), (0, 1), (0, 2), (2, 2), (2, 1)
Move de Pion Noir (1, 6),	(1, 5)	(2, 5), (2, 4), (3, 4), (1, 3), (0, 5), (1, 7)
new Move(new Point(2, 1), 3) de (0, 0)	(2, 1), (4, 2), (6, 3)	
new Move(new Point(0, 1), 1)	(0, 2),(0, 3),(0, 4)	
Move de Knight Blanc (6,0)	(7, 2)	(7, 1), (7, 3)

Table 2: Tests CastleMove()

Nom	**OK**
Test du Petit Roque sur roi blanc	OK
Test du Grand Roque sur roi blanc	OK
Test du Petit Roque sur roi noir	OK
Test du Grand Roque sur roi noir	OK
Test roque avec des pions qui bloquent le chemin ->invalide	OK
Test roque ne se produit pas quand le roi a déjà bougé	OK
Test roque ne se produit pas quand la tour a déjà bougé	OK
Test grand roque blanc ne se produit pas si on tente de bouger le roi de 3 cases	OK
Test de faire un roque avec le roi en échec ->invalide	OK
Test de faire un roque avec la case intermédiaire menaçée ->invalide	OK
Test de faire un roque alors que le roi est menacé sur la case d'arrivée ->invalide	OK

Table 3: Tests EnPassant()

Nom	**OK**
En passant à droite avec pion attaquant blanc	OK
En passant à gauche avec pion attaquant blanc	OK
En passant à droite avec pion attaquant noir	OK
En passant à gauche avec pion attaquant noir	OK
En passant avec la victime qui a bougé d'une case ->invalide	OK
En passant avec l'attaquant qui n'a pas fait son En passant tout de suite. ->invalide	OK

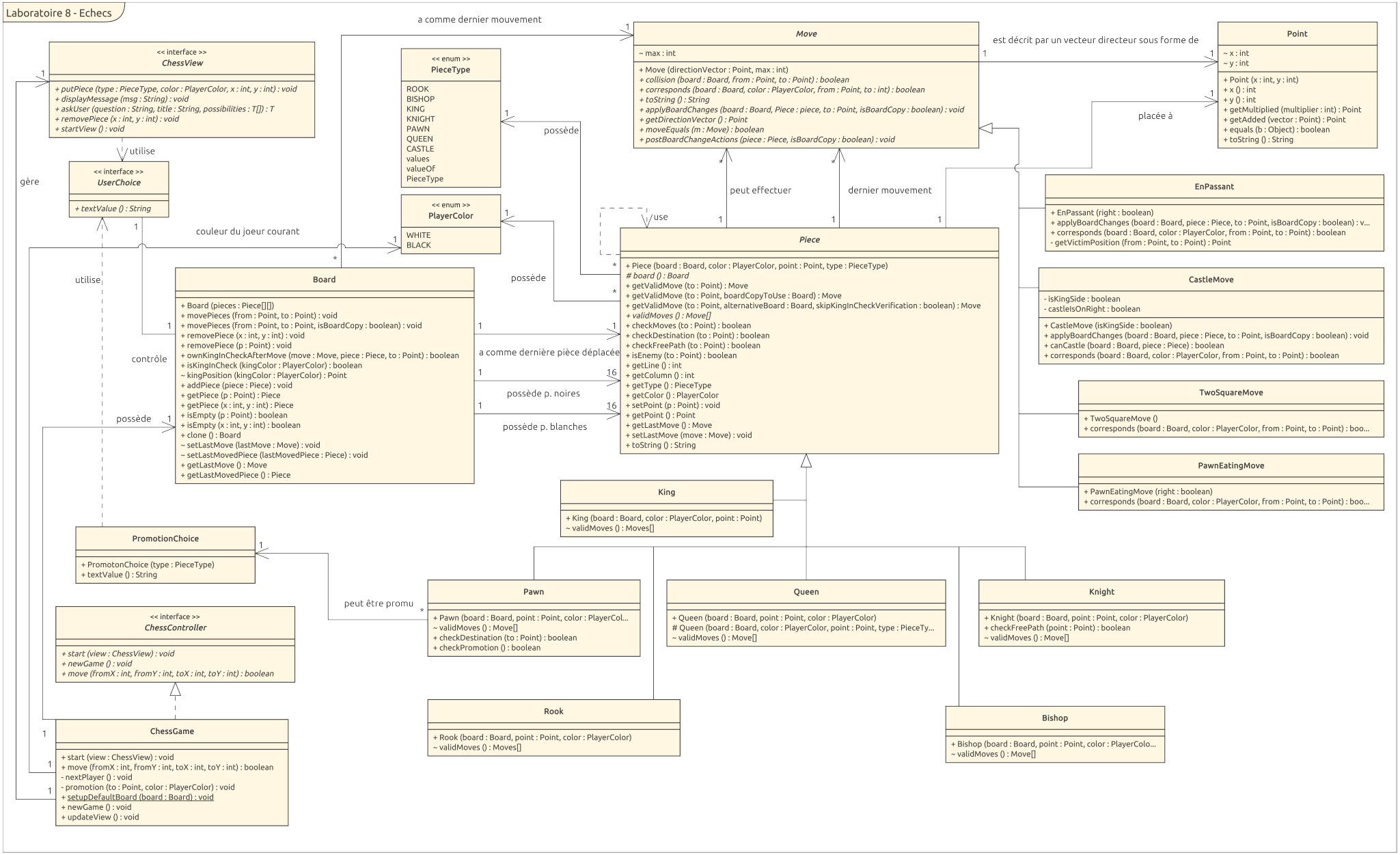
Table 4: Tests KingInCheck()

Nom	**OK**
Les positions des 2 rois peuvent être trouvées	OK
Les positions des 2 rois sont safe sur le plateau par défaut	OK
Le roi noir est en échecs quand la dame blanche le menace	OK
Les 2 rois peuvent être en échecs en même temps	OK
On ne peut pas faire un mouvement si cela met notre roi en échec	OK

Table 5: Tests Move()

Nom	**OK**
Le chemin libre d'un pion est bien checké	OK
Le cavalier a toujours le chemin libre	OK
La position interne de la pièce est modifiée après le mouvement	OK
Test que la position interne n'est pas modifiée si le mouvement est annulé (car roi en échecs)	OK
Le pion peut avancer de 1 case en avant et de 2 cases seulement la première fois.	OK
Le fou blanc en bas à gauche peut bouger en diagonale à droite et à gauche si pas de pion sur le chemin.	OK
La tour blanche peut bouger en avant de 1 case ou plus en avant si pas de pion devant	OK

5 Diagramme UML



6 Conclusion

En conclusion, notre objectif a été de créer un jeu d'échecs en essayant d'être le plus optimisé et complet possible, tout en intégrant les règles de bases. La construction de nos classes a été pensée pour permettre le plus de flexibilité possible.

Si on imaginait un instant de devoir définir une nouvelle pièce troubadour se déplaçant de 2 cases à gauche ou à droite (une fois sur deux de chaque côté), et qui mange toutes les pièces qu'il touche, il suffirait de créer une classe `Troubadour` héritant de `Piece`, de créer un mouvement particulier `StrangeMove` héritant de `Move` permettant de détecter la valider que le mouvement droite ou gauche est valide en fonction du dernier mouvement. Une fois le mouvement validé, il reste à implémenter dans `StrangeMove.applyBoardChanges()` les modifications du plateau à faire en plus du déplacement du troubadourd, c'est à dire de supprimer les pièces qui auraient touchées.

Finalement, nous avons essayé de garantir que notre implémentation puisse s'adapter à certaines modifications de règles restant dans l'esprit du jeu.