Laboratoire 8 - Chess

Leonard Cseres, Aladin Iseni

8 janvier 2025

Table des matières

1 Introduction

L'objectif de ce laboratoire est de développer un jeu d'échecs fonctionnel respectant les règles de base. Le projet inclut les fonctionnalités suivantes: déplacements des pièces, coups spéciaux (roque, prise en passant, promotion des pions) et gestion des états de jeu (par exemple, échec). Les objectifs bonus consistent à implémenter la détection de l'échec et mat ainsi que du pat.

Pour simplifier le développement, les éléments suivants nous ont été fournis:

- **Enums:** `PieceType` pour les types de pièces et `PlayerColor` pour les couleurs des joueurs.
- **Interfaces:** `ChessController` et `ChessView` pour la gestion du jeu et de l'interface utilisateur.
- **Vues pré-construites:** Une vue graphique (`GUIView`) et une vue en mode texte (`ConsoleView`).

L'implémentation se concentre sur un nouveau package `engine` qui encapsule la logique du jeu tout en exploitant les interfaces fournies pour l'interaction.

Le rendu est composé de 2 versions:

- Sans modifications au package `chess` fourni
- Avec modifications au package `chess` fourni (c.f. Extensions)

---

2 Conception et Architecture

Notre approche respecte les principes de conception orientée objet, en garantissant l'encapsulation, la réutilisabilité et la modularité. Le package `engine` contient les classes et la logique pour la gestion du jeu, le suivi de l'état de l'échiquier et la génération des mouvements.

2.1 Structure

Comme mentionné précédemment, le notre implémentation se situe dans le package `engine`.

- `app` ..... point d'entrée de l'application
- `chess` ..... *code founi*
- `engine` ..... notre implémentation
  - `board` ..... logique de l'échiquier
  - `generator` ..... génération de mouvements possibles pour les pièces
  - `move` ..... gestion des différents mouvements
  - `piece` ..... logique des pièces d'échec

2.2 Composants Clés

- **`ChessEngine`:** Gère le déroulement du jeu et communique avec le contrôleur de l'échiquier.

- **`ChessBoardContoller`:** Expose l'échiquier en contrôlant la view (`ChessView`).
- **`ChessBoard`:** Représente l'échiquier, suit les pièces et valide les états du jeu.
- **`ChessBoardReader/ChessBoardWriter`:** Interface de lecture/écriture de l'échiquier.
- **`ChessPiece`:** Classe abstraite définissant le comportement commun à toutes les pièces, étendue par des sous-classes spécifiques.
- **`MoveGenerator`:** Classe abstraite responsable de la génération des mouvements possibles pour les pièces.
- **`ChessMove`:** Représente un type de mouvement aux échecs.

2.3 Détails de Conception

Séparation `ChessBoard` et `ChessBoardContoller`

Nous avons découplé la logique de l'échiquier avec la mise à jour de la vue (`ChessView`) afin de pouvoir cloner l'échiquier sans être lié à la vue.

Cela nous permet d'exécuter des mouvements sur la classe `ChessBoardContoller` pour mettre à jour l'interface et exécuter des mouvements sur la classe `ChessBoard` sans mettre à jour l'interface.

Interfaces `ChessBoardReader` et `ChessBoardWriter`

Pour encapsuler l'échiquier, deux interfaces limitent l'accès de `ChessBoard`.

Par exemple, une pièce d'échec est uniquement intéressée à lire l'état de l'échiquier pour générer différents types de mouvements. Alors qu'un mouvement (`ChessMove`) doit pouvoir modifier l'état de l'échiquier.

Classe Abstraite `PromotableChessPiece`

Le rôle de cette classe est d'implémenter l'interface `UserChoice` fournie afin d'avoir un représentation textuelle pour l'utilisateur de l'interface de la pièce lors d'une promotion.

2.4 Diagramme UML

Le diagramme UML fournit une vue d'ensemble de la structure et des relations du système. Les éléments grisés représentent le code que nous avons utilisé et non pas implémenté.

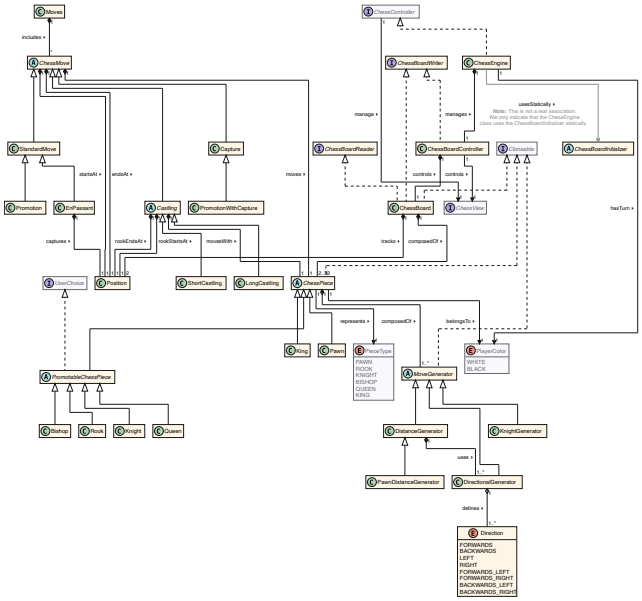

Figure 1: Schéma UML (Vue simplifiée)

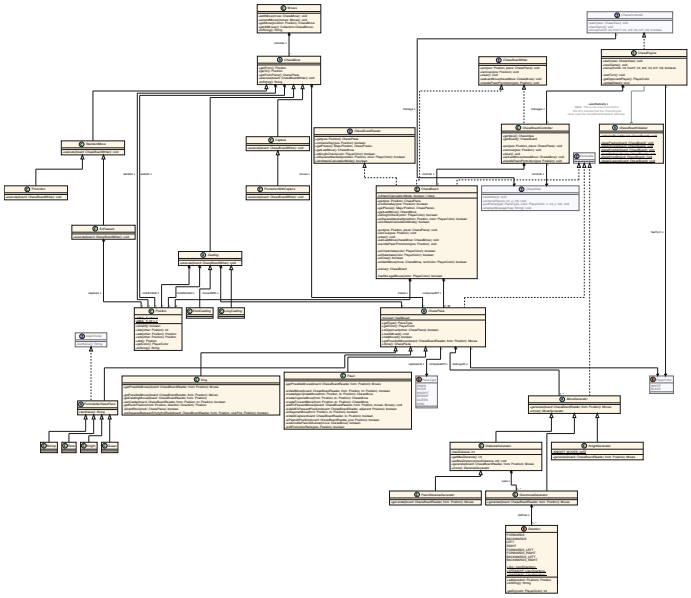

Figure 2: Schéma UML (Vue détaillée)

3 Caractéristiques Principales

Notre jeu implémente les fonctionnalités suivantes pour une partie de joueur contre joueur:

- Les mouvements de base des pièces (pion, tour, cavalier, fou, reine, roi).
- La capture des pièces adverses.
- Le petit et le grand roque
- La prise en passant
- La promotion des pions en avançant
- La promotion des pions en capturant
- La détection et l'affichage de:
  - L'échec
  - L'échec et mat
  - Le pat
  - L'impossibilité de mater

3.1 Règles Spéciales

- **Roque:** Vérifie que le roi et la tour concernés n'ont pas bougé, que le chemin est libre et que les cases traversées ne sont pas attaquées.
- **Prise en passant:** Implémente la capture d'un pion adjacent qui a avancé de deux cases à son premier mouvement.
- **Promotion de pions:** Demande au joueur de choisir un type de promotion (tour, cavalier, fou ou dame).

---

4 Tests Effectués

Tests effectués	**Résultat**
Mettre le roi blanc en échec où le seule mouvement possible est l'attaque de la pièce blanche par une pièce noire	Vrai
En Passant est uniquement pratiquable lorsque le pion adverse avance de deux cases	Vrai
En passant est praticable uniquement au tour suivant et pas 2 tours après	Vrai
Le roque est uniquement praticable si le roi et la tour en question n'ont pas bougé	Vrai
Le roque est pratiquable uniquement si les cases sur lesquelles passe le roi ne sont pas attaquées	Vrai
Les pions peuvent avancer de deux cases uniquement lors de leur premier déplacement	Vrai
Chaque pièce avance dans la bonne direction	Vrai
Uniquement les chevaux peuvent sauter des pièces	Vrai
Les pièces ne peuvent pas découvrir un échec	Vrai
Le roi ne peut pas se mettre en échec	Vrai
Lorsque le roi est en échec, uniquement les mouvements de défenses sont praticables	Vrai

Tests effectués	Résultat
Une pièce ne peut que capturer les pièces d'une autre couleur	Vrai
Un pion peut être promu en reine, fou, chevalier ou tour	Vrai
Un message Check s'affiche lorsque le roi est en échec et Checkmate lorsque quelqu'un a gagné	Vrai
Un message Draw s'affiche lorsqu'il n'est plus possible de faire un checkmate avec le matériel restant	Vrai

4.1 Défense par l'attaque

Les images suivantes montrent que le joueur blanc est obligé d'attaque le fou en H4 avec le cavalier en F3 afin de défendre son roi.

Figure 3: Le roi est bloqué

Figure 4: Le cavalier peut défendre le roi

4.2 Checkmate

Cette image montre que notre jeu est capable de détecter un échec et mat.

Figure 5: Checkmate

4.3 Stalemate

Cette image montre que notre jeu est capable de détecter un pat.

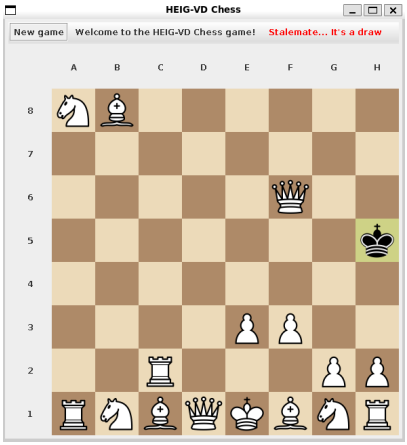

Figure 6: Stalemate

4.4 Draw

Nous observons sur l'image suivante le message d'égalité dû au manque de pièces pour effectuer un échec et mat.

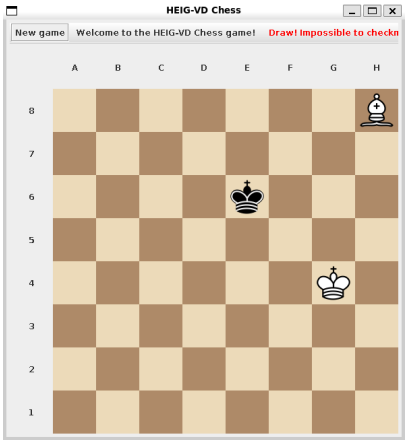

Figure 7: Draw

4.5 En Passant

Sur les deux images ci-dessous, nous pouvons observer que notre jeu propose l'attaque En Passant et permet de l'exécuter.

Figure 8: Le pion a la possibilité de capturer En Passant

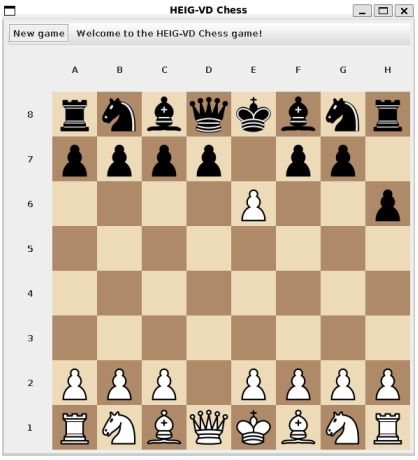

Figure 9: Le pion capture En Passant

4.6 Castling

Les trois images suivantes montrent qu'il n'est pas possible d'effectuer un castling si les cases du passage du roi sont attaquées.

Figure 10: Le roi est bloqué car les cases sont attaquées

Figure 11: Le roi peut effectuer un roque

Figure 12: Le roi effectue un roque

4.7 Promotion

Ci-dessous, nous observons qu'il est possible de promouvoir un pion en reine, tour, fou ou cavalier à l'aide d'une fenêtre de sélection.

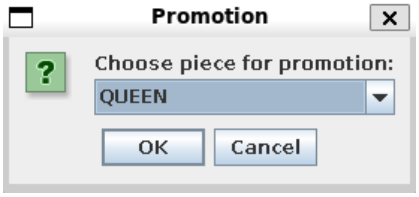

Figure 13: Le pion est prêt à être promu

Figure 14: Le joueur peut choisir la promotion du pion

Figure 15: Chaque promotion

---

5 Extensions

5.1 Génération des Mouvements

La hiérarchie `MoveGenerator` encapsule la logique de génération des mouvements:

- **`DirectionalGenerator`:** Pour les mouvement directionnels.
- **`DistanceGenerator`:** Gère les mouvements avec des portées variables, comme les pions.
- **`KnightGenerator`:** Pour les mouvements en L propres aux cavaliers.
- **`PawnDistanceGenerator`:** Spécialise `DistanceGenerator` pour autoriser le mouvement de 2 cases quand le pion n'as pas encore bougé, puis 1 seule case.

Au point de vue conception, la classe `DistanceGenerator` peut utiliser une liste de `DirectionalGenerator` pour obtenir les mouvements d'une distance maximum avec de la directionnalité.

Par exemple, le roi, peut se déplacer dans tous les sens mais d'une seule case à la fois.

5.2 Gestion des États de Jeu

La logique de détection des états de jeu est encapsulée dans la classe `ChessBoard`. Chaque cas est vérifié de la manière suivante:

- **`Échec et Mat`:** Vérifie si le roi est en échec et qu'il n'a aucun mouvement légal disponible.
- **`Pat`:** Vérifie que le roi n'est pas en échec et qu'aucun mouvement légal n'est disponible.
- **`Impossibilité de mater`:** Vérifie qu'il n'y a plus de matériel nécessaire pour mater. C'est-à-dire, qu'il vérifie si une des 4 situations suivantes est

vraie:
- Les deux joueurs n'ont plus que leur roi.
- L'un des joueurs n'a plus que son roi et l'autre joueur n'a plus que son roi et un fou.
- L'un des joueurs n'a plus que son roi et l'autre joueur n'a plus que son roi et un cavalier.
- Chaque joueur a un roi et un fou, mais les deux fous sont sur des cases de la même couleur.

5.3 Package `chess`

Nous avons aussi apporté quelques modifications au package `chess` fourni. Comme mentionné dans l'introduction, nous avons fourni deux versions qui incluent/excluent ces modifications.

Changements Esthétiques

Nous avons modifier les images des pions ainsi que les couleurs de l'échiquier. Les images sont sous licence AGPL-3.0 et proviennent du projet Lichess[1].

Soulignement des Mouvements Possibles

Lorsqu'un utilisateur clique sur une pièce, les cases où la pièce peut bouger sont surlignées. Cette fonctionnalité nous a aidé a déboguer les mouvements de pièces.

Pour implémenter cela, nous avons rajouté la méthode suivante dans `ChessView`:

```
/**
 * Highlights with a green dot the given positions
 *
 * @param pos the list of positions
 */
void highlightPositions(List<Position> pos);
```

Et nous avons rajouté la méthode suivante dans `ChessController`:

```
/**
 * Called when the user selects a piece on the board
 *
 * @param x the piece x position
 * @param y the piece y position
 */
void select(int x, int y);
```

Cela nous permet de réagir aux évènements 'select' afin de souligner les mouvements possible.

[1] https://github.com/lichess-org/lila/tree/master/public/piece/cburnett

6 Conclusion

Ce projet a renforcé les principes de programmation orientée objet tout en abordant des règles et interactions complexes. Les défis ont inclus:

- Garantir l'encapsulation tout en gérant les comportements variés des pièces.
- Traiter les cas limites dans les coups spéciaux et les conditions de fin de jeu.

Améliorations futures possibles:

- Ajouter une IA pour un mode solo.
- Proposer des suggestions de mouvements pour améliorer l'expérience utilisateur.

A Annexes

A.1 Listing Java

c.f. page suivante.