# POO Labo 8

## Structure des pièces d’échecs

L'implémentation du jeu repose sur une structure centrée autour de la super-classe ChessPiece. Cette classe représente les caractéristiques communes à toutes les pièces d’échecs et contient des attributs tels que le type de la pièce, défini par une énumération (enum PieceType), sa couleur (enum PlayerColor), la liste des mouvements possibles et sa position actuelle sur le plateau. Ces caractéristiques assurent une base commune et cohérente, tout en permettant à chaque type de pièce de personnaliser son comportement en héritant de cette super-classe. L'interface PromotablePiece, dédiée aux pions, permet de gérer les actions spécifiques comme leur promotion lorsqu’ils atteignent l’autre côté du plateau.

## Gestion des règles et mouvements

La classe GameManager est au cœur de la gestion des règles et des mouvements du jeu. Elle commence par positionner les pièces sur le plateau lors de l'initialisation. Pendant la partie, elle vérifie si les mouvements proposés par les joueurs respectent les règles propres à chaque pièce et s’assure qu’ils ne mettent pas le joueur en situation d’échec. Une fois validé, le mouvement est appliqué sur le plateau. Si une pièce adverse est capturée, elle est retirée du jeu. En plus des mouvements standards, GameManager gère des cas particuliers comme la promotion des pions. Lorsque l’un d’eux atteint l’autre côté du plateau, le joueur peut choisir de le transformer en tour, cavalier, fou ou dame. Les roques, petits et grands, sont également implémentés. Ces mouvements permettent de déplacer le roi de deux cases en respectant des conditions spécifiques, comme l’absence d’échec ou l’immobilité préalable du roi et de la tour. La prise en passant est également prise en charge, permettant à un pion de capturer un autre pion ayant avancé de deux cases au tour précédent.

## Interface utilisateur et affichage

L’interface utilisateur du jeu est proposée sous deux formes : graphique ou en mode console. Ces deux versions permettent aux joueurs d’interagir avec le programme de manière intuitive. Lorsqu’un événement particulier survient, comme un roi mis en échec, des messages informatifs, tels que "Check!", sont affichés à l’utilisateur via la méthode displayMessage(String). Ce système assure une expérience fluide tout en tenant l'utilisateur informé des situations importantes du jeu.

## Règles implémentées et limitations

L’implémentation respecte les règles fondamentales du jeu d’échecs, y compris les mouvements des pièces, les roques, la prise en passant et la promotion des pions. Cependant, certaines fonctionnalités avancées, comme la détection de l’échec et mat ou des situations de match nul (par pat ou impossibilité de mater), ne fonctionnent pas. Le programme repose sur le principe que l'utilisateur joue pour les deux camps et ne propose pas d’adversaire automatisé ni d’affichage des mouvements possibles des pièces. Cela laisse aux joueurs la responsabilité de connaitre les coups à jouer.

## Bonnes pratiques et organisation

L’implémentation suit des principes stricts d’encapsulation et de modularité. Les énumérations PieceType et PlayerColor sont utilisées pour représenter des données de manière claire, sans influencer directement les règles du jeu. L’ensemble du code spécifique est isolé dans un package dédié « engine », afin de préserver la clarté et la maintenabilité de l’architecture. Par ailleurs, les classes fournies dans le package de base chess ne sont pas modifiées, ce qui clarifie la séparation entre le code de départ et celui ajouté pour les fonctionnalités. Cette approche méthodique assure une extensibilité et une robustesse idéales pour le projet.

## Choix et hypothèses de travail

### Paquetage piece

Le paquetage piece regroupe toutes les classes représentant les différentes pièces du jeu d'échecs. Il inclut également des interfaces modélisant des comportements spécifiques pour certaines situations.

#### ChessPiece

ChessPiece est la classe mère de toutes les pièces. Elle contient les attributs et méthodes communs, tels que :

* Le type de pièce, la couleur, la position et les mouvements possibles.
* Les méthodes publiques suivantes :
  + canMoveTo() : Vérifie si un déplacement est autorisé.
  + move() : Met à jour la position de la pièce.
  + isAttacking() : Vérifie si une pièce peut attaquer une position donnée.
  + isSameColor() : Indique si une pièce est de la même couleur qu'une autre.

Les classes dérivées redéfinissent certaines méthodes comme possibleLengthDistance() pour gérer les déplacements spécifiques.

#### SpecialFirstMovePiece

SpecialFirstMovePiece est une classe utilisée pour les pièces avec un comportement spécifique lors de leur premier déplacement. Deux interfaces associées :

* CastlingPiece : Gère le roque.
* PromotablePiece : Permet la promotion (utilisée par le pion).

#### Pawn

Le pion redéfinit :

* canReach() : Intègre les règles spécifiques comme les captures en diagonale ou la prise en passant.
* isAttacking() : Prend en compte les attaques diagonales.
* canPromote() : Vérifie si un pion peut être promu.

#### King

Le roi redéfinit :

* canReach() : S’assure qu’il ne se déplace pas sur une case sous échec.
* isAttacking() : Gère les attaques en fonction des règles spécifiques.
* canMoveTo() : Intègre les conditions pour le roque.

### Paquetage util

Ce paquetage regroupe des classes utilitaires essentielles au fonctionnement du jeu.

#### Direction

Direction est un type énuméré définissant les vecteurs de déplacement possibles pour les pièces. Elle inclut :

* 8 directions linéaires unitaires.
* 8 directions pour les mouvements en "L" (comme ceux du cavalier).

Les méthodes principales :

* offSetOf() : Calcule une direction entre deux points.
* opposite() : Donne la direction opposée.
* isDiagonal() : Vérifie si une direction est diagonale.

#### Point

Point est un record contenant deux entiers x et y, représentant une position. Il inclut une méthode pour calculer un nouveau point en fonction d’une direction donnée.

#### ChessString

Cette classe finale regroupe des chaînes de caractères utilisées dans le jeu pour centraliser et faciliter leur gestion.

#### Assertions

Assertions propose des méthodes statiques comme :

* assertTrue() : Vérifie qu'une condition est respectée.
* assertNotNull() : Vérifie qu'un objet n'est pas null.

En cas d'échec, une exception IllegalArgumentException est levée.

#### Board

Board est une classe finale représentant l’état du plateau. Ses fonctionnalités incluent :

* Un tableau 8x8 de pièces (pieces).
* Une liste des pièces promouvables et des deux rois.
* La méthode fillBoard() : Initialise le plateau avec les positions de départ.
* La méthode reset() : Réinitialise la partie.

Les principales méthodes :

* addPiece(), removePiece() : Ajout ou suppression de pièces.
* getPiece(), isOccupied() : Informations sur une position donnée.

#### GameManager

Le GameManager est le contrôleur central du jeu. Il implémente l’interface ChessController, avec les méthodes :

* newGame() : Prépare une nouvelle partie.
* start() : Initialise la vue du jeu (ChessView).
* move() : Réalise un déplacement.

Attributs

* board : Instance de Board pour gérer l’état du jeu.
* view : Instance de ChessView pour gérer l'affichage.
* turn : Entier indiquant le tour actuel.
* mandatoryAdversaryMoves : Une HashMap des mouvements obligatoires en cas d’échec.

Fonctionnalités principales

1. Gestion des déplacements :

* movePreconditions() : Vérifie la validité d’un mouvement.
* executeSpecialMoves() : Gère des mouvements spéciaux comme le roque ou la prise en passant.
* movePiece() : Effectue un déplacement et met à jour l’état du jeu.

1. Actions post-déplacement :

* promoteIfNeeded() : Vérifie si une pièce peut être promue.
* postMoveActions() : Détecte les fins de jeu (pat, échec et mat, etc.).

1. Gestion des fins de partie :

* postGameActions() : Permet de redémarrer une partie.
* clearView() : Réinitialise la vue après une partie.

## Conclusion

Le développement de ce jeu d’échecs nous a permis d’exploiter au maximum les concepts fondamentaux de la programmation orientée objet. L’utilisation d’interfaces pour modéliser des comportements spécifiques et de classes abstraites pour mutualiser des fonctionnalités communes a été au cœur de notre conception.

La redéfinition des méthodes dans les classes dérivées a simplifié le code tout en permettant de respecter les règles complexes du jeu. Nous avons aussi optimisé les différentes mécaniques du jeu afin de garantir un code non seulement fonctionnel, mais également performant.

Bien que le jeu soit pleinement opérationnel, malgré certains problèmes au niveau de la détection de l’échec et mat, des améliorations restent possibles. Cependant, grâce à la conception modulaire et extensible, ces évolutions pourront être intégrées facilement, garantissant une base de code robuste et maintenable à long terme.