13/01/2022

Valentin Kaelin et Jonathan Friedli

Rapport Laboratoire 8 : Echecs

Table des matières

[Introduction 1](#_Toc92214104)

[Diagramme des classes 2](#_Toc92214105)

[Choix de modélisation et d’implémentation 3](#_Toc92214106)

[GameManager 3](#_Toc92214107)

[Board 3](#_Toc92214108)

[Système d’événements 3](#_Toc92214109)

[Stockage des pièces 3](#_Toc92214110)

[Liste de rois 3](#_Toc92214111)

[Compter les tours 3](#_Toc92214112)

[Cell 4](#_Toc92214113)

[Moves 4](#_Toc92214114)

[Piece 4](#_Toc92214115)

[Interface UserChoice 4](#_Toc92214116)

[postUpdate() 4](#_Toc92214117)

[Déplacements spéciaux 4](#_Toc92214118)

[FirstSpecificMove 4](#_Toc92214119)

[Pawn 4](#_Toc92214120)

[Tests effectués 4](#_Toc92214121)

[Tests généraux 5](#_Toc92214122)

[Tests de la Reine 5](#_Toc92214123)

[Tests du Roi 5](#_Toc92214124)

[Tests de la Tour 6](#_Toc92214125)

[Tests du Fou 6](#_Toc92214126)

[Tests du Cavalier 6](#_Toc92214127)

[Tests du Pion 7](#_Toc92214128)

# Introduction

Dans le cadre de ce laboratoire n°8, nous avons dû implémenter un programme permettant de jouer aux échecs. Deux interfaces nous étaient fournies (une interface graphique ainsi qu’un mode console) et notre but a été de réaliser toute la partie logique du jeu. Pour finir, les implémentations de l’échec et mat et des pat n’ont pas été réalisées.

# Diagramme des classes

**A AJOUTER UNE FOIS FINI**

# Choix de modélisation et d’implémentation

## GameManager

Nous avons séparé la gestion de la communication avec la vue du véritable plateau pour plusieurs raisons. Tout d’abord, cela nous a permis de bien séparer les différents contextes. En effet, la classe *GameManager* implémentant l’interface *ChessController* est la seule à interagir avec la vue et les autres classes communiquent avec elle via un système d’évènements sur lequel nous reviendrons en détails plus tard. De plus, cela a été plus cohérent afin de transmettre aux différentes pièce le *Board* sur lequel elles se trouvent e au lieu de leur passer une classe contenant plus de fonctionnalités et ayant moins de rapport avec elles.

La vérification qu’un roi est en échec est réalisée dans la classe *GameManager* car il aurait été compliqué de lancer un événement aux multiples endroit où la mise en échec est vérifiée. L’inconvénient de cette pratique est que la vérification est réalisée à chaque fin de tour et est plutôt coûteuse (O(n2) sur le tableau des pièces).

## Board

### Système d’événements

La classe *Board* émet des évènements qu’écoute la classe *GameManager* dans plusieurs cas :

* Une pièce a été supprimée d’une case.
* Une pièce a été ajoutée sur une case.
* La promotion d’un pion est demandée.

Cela nous permet d’avoir une interface graphique à jour sans devoir manuellement ajouter/supprimer les pièces de la vue à chaque modification du plateau virtuel.

Les interfaces des Listeners ont été définies comme internes à la classe *Board* car il est peu judicieux de les ajouter chacune dans un fichier séparé en perdant le contexte qu’elles sont liées à la classe *Board*.

Stockage des pièces  
Les pièces sont stockés dans un simple tableau à deux dimensions afin d’y accéder en O(1) lors des recherches. Il aurait également été possible de faire une HashMap mais cela nous a semblé compliqué inutilement l’implémentation étant donné que les états des pièces ne sont pas forcément fixes.

### Liste de rois

En plus du tableau à deux dimensions, une ArrayList contenant les deux rois a été créée afin de faciliter la mise en échec. En effet, cela nous évite de devoir parcours le double tableau à la recherche du roi qui nous intéresse.

### Compter les tours

Nous avons un compteur s’incrémentant après chaque tour afin de récupérer le joueur actuel ainsi que pour faciliter la gestion d’un des cas de nulle pour une future implémentation (50 coups sans avance de pion ni de prise). Ce compteur est également utile lors de l’implémentation de la prise en passant.

## Cell

Une classe représentant une case du plateau a été réalisée afin d’offrir plus de possibilités. Il aurait été possible de stocker directement les deux coordonnées x et y dans la pièce mais cela nous aurait embêtés dans les actions suivantes :

* Réaliser des opérations mathématiques entre deux cases, afin de par exemple itérer sur toutes les cases d’un déplacement.
* Comparer facilement 2 positions.
* Vérifier si une case est accessible depuis une autre.

## Moves

De nombreux constructeurs ont été mis à disposition afin d’éviter à l’utilisateur de la classe de devoir mettre des valeurs par défaut qui n’ont pas énormément de sens.

La classe *LinearMove* nous a permis de gérer à la fois les mouvements horizontaux et verticaux, les diagonales et les déplacements du cavalier grâce à l’utilisation des diverses méthodes de la classe *Cell*.

## Piece

### Interface UserChoice

Les pièces implémentent l’interface *UserChoice* de la vue afin de faciliter la gestion du choix de la pièce lors de la promotion. Le point négatif de cette approche est qu’on lie la vue à l’implémentation logique alors qu’on avait par avant bien réussi à faire la distinction des deux. Il aurait sûrement été possible de créer des types d’abstraction au-dessus des pièces lors de la promotion mais cela nous a encore une fois semblé se compliquer beaucoup la vie.

### postUpdate()

La méthode n’est pas abstraite car elle n’est pas redéfinie dans toutes les pièces. Il est donc plus facile de laisser simplement son corps vide.

### Déplacements spéciaux

Les déplacements spéciaux tels que les roques ou la prise en passant sont directement implémentés dans la pièce spécifique. Il aurait été possible de créer des classes à part comme pour les autres mouvements mais comme ceux-ci sont utilisés dans une seule et unique classe, nous sommes allés droit au but.

## FirstSpecificMove

Une deuxième couche de classe abstraite a été implémentée afin de pouvoir gérer les 3 types de pièces pour lesquelles le premier déplacement nous est important (roi, pion, tour). Cela nous a permis d’éviter d’avoir un attribut boolean en plus dans les 7 classes ou avoir à répéter cet attribut dans les 3 classes citées ci-dessus.

## Pawn

C’est le pion lui-même qui vérifie s’il peut être promu dans la méthode *postUpdate()* afin de ne pas avoir à déléguer cette tâche. Par conséquent, un accesseur au listener de la classe *Board* a dû être implémenté.

# Tests effectués

Nous allons lister par la suite les différents tests que nous avons réalisés, en plus d’avoir joué quelques parties complètes.

## Tests généraux

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests de la Reine

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests du Roi

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests de la Tour

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests du Fou

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests du Cavalier

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |

## Tests du Pion

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
|  |  |
|  |  |
|  |  |
|  |  |