Blin Sébastien

École supérieure d'ingénieurs de Rennes

2ème année
Parcours Informatique

Projet MDI

Réalisation d'un jeu d'échecs

Sous l'encadrement de :
Acher Mathieu

1 Introduction

Le but de ce projet était de modifier un programme existant (un jeu d'échecs) et de lui apporter des fonctionnalités. Le code souffrant de défauts de conception. Ce projet a de plus permis la mise en place de différents design patterns vu en cours. Le code de ce projet est disponible ici : `https://github.com/AmarOk1412/ChessMDI/`.

2 Au commencement

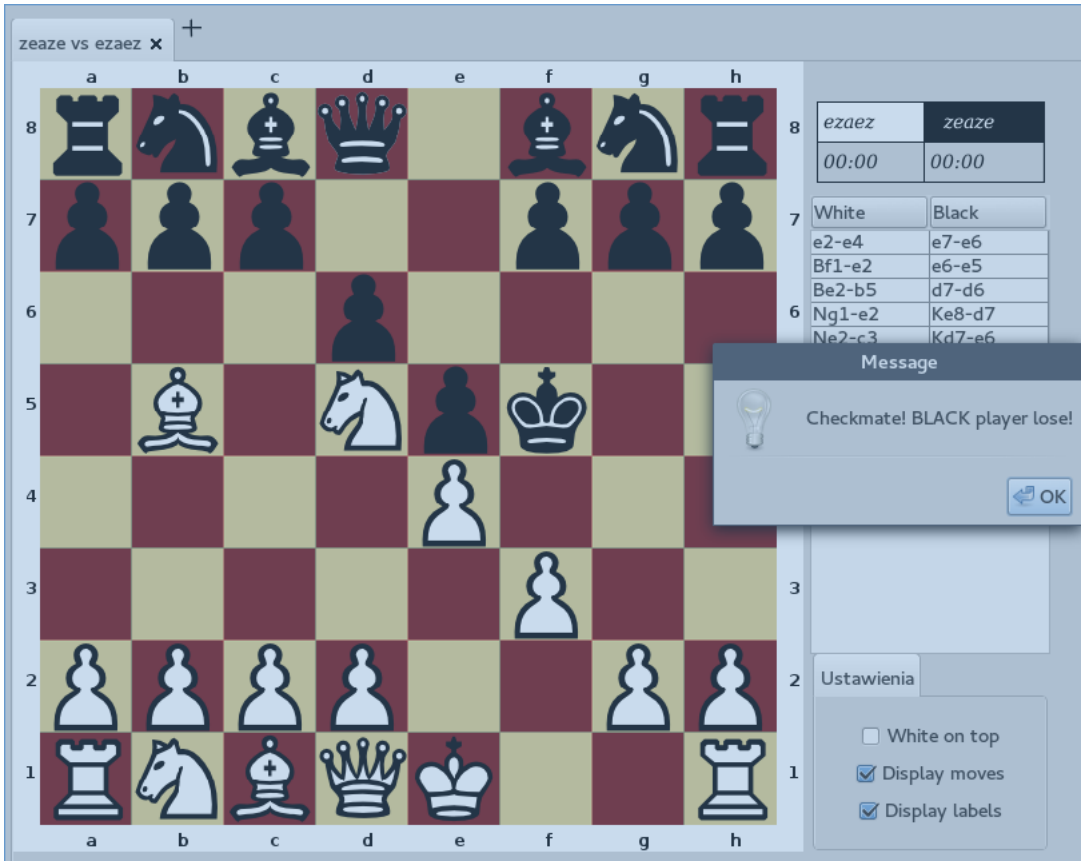
Le diagramme UML du projet étant plutôt massif, il ne sera pas inclu pour des soucis de lisibilité directement dans ce rapport. Vous pouvez le trouver à l'url suivante : `https://github.com/AmarOk1412/ChessMDI/blob/master/uml/diagramme.png?raw=true`.
Dans ce projet, plusieurs design pattern sont déjà en place. On peut noter la présence d'un design pattern *Template* pour les *Behavior*, ce qui permet d'avoir la même fonctionnalité dans toute les sous classes (ici *getSquaresInRange()*). Ce design pattern est adapté lorsqu'on souhaite traiter tout un groupe d'objet de la même manière (ici comme des comportements). Il est composé de 2 classes. Une classe abstraite (ici *Behavior*) avec une méthode (par exemple *getSquaresInRange()*), et d'une classe concrète qui en hérite (ici les différentes classes héritant de *Behavior*). Un design pattern de type *Template* est présent pour les différentes classes héritant de Piece. Ce qui permet de faire la bonne action en fonction de la pièce.
Un design pattern de type *Composite* est présent pour les éléments graphiques. Ce design pattern est composé de 3 classes. Une classe component, élément de base abstrait ; une classe composite qui est une classe pouvant contenir des autres components (par exemple ici *javax.swing.GroupLayout* ; et une classe feuille, qui ne peuvent pas contenir d'autres components même si à priori ici, toutes les classes peuvent contenir des components (*JLabel, JProgressBar, etc*).
De plus, des design pattern *Observer* sont mis en place pour les différentes interactions avec l'interface. Cette mis en place se fait via des fonctions comme *addActionListener* qui activent des évènements. Ce design pattern est composé d'une classe *Observer* (interne à swing ici) et de sujets qui s'incrivent a des évènements (ici *JChessView* par exemple). Parfois, un patron *Command* aurait été plus adapté, mais il s'agit de l'implémentation de la librairie graphique de base.
Enfin un design pattern *Singleton* est présent pour la classe *Pieces2D*. Ce design pattern consiste a rendre le constructeur privé afin d'éviter la création de multiples instances et avoir une fonction pour n'obtenir qu'une seule instance d'une classe. Dans ce code il aurait pu être utilisé plus souvent (par exemple pour la classe *JChessView*).

3 Bugs de déplacement

Plusieurs bugs de déplacement ont été trouvé dans le projet. Le premier concerne le fou qui ne pouvait pas se déplacer sur une diagonale (ainsi que la reine qui utilisait la même Behavior). Il s'agissait d'une ligne commentée :

```
//list.addAll(getMovesForDirection(DIRECTION_RIGHT,  DIRECTION_UP)); //right-up
```

Un second bug dans le déplacement est le fait que le roi pouvait se déplacer dans une position où il était en échec tout en devenant échec et mat au second tour, l'erreur venant de *willBeSafeAfterMove()*.



Des cas de tests ont été ajoutés dans le fichier de test pour vérifier le bon déplacement de toutes les pièces.

4 Monteur pour la visibilité

Dans le package *jchess.core.Moves*, j'ai décidé de mettre en place un pattern design *Builder* couplé avec un design pattern *Strategy*. Le premier me permet de construire un objet par chaînage. Ainsi les méthodes from/to retournent une instance de ce même objet, comme par exemple :

```
        public AlgebricChainMove from(String value)
{
          if(value.length() != 2)
          {
                  LOG.error("from value incorrecte : " + value );
                  return this;
           }
           int index = _horizontal.indexOf(value.getBytes()[0]);
          this._xFrom = index;
           index = _vertical.indexOf(value.getBytes()[1]);
           this._yFrom = index;
           return this;
}
```

Ce qui donne en exemple concret :

```
new AlgebricChainMove(board).from("d2").to("d4").move();
```

De plus, le second pattern permet de choisir la solution d'écriture de mouvement que l'on souhaite. On a le choix entre 2 méthodes. *AlgebricChainMove* et *NumericChainMove*. Les 2 solutions utilisent la méthode move définie dans la classe abstraite *MoveBuilder*. Ces 2 classes créées ne sont utilisées que pour les tests dans mon projet. En effet, ces classes rendent les tests beaucoup plus lisibles et simple d'écriture. Mais elles sont peu adaptées pour l'utilisation dans le reste du code.

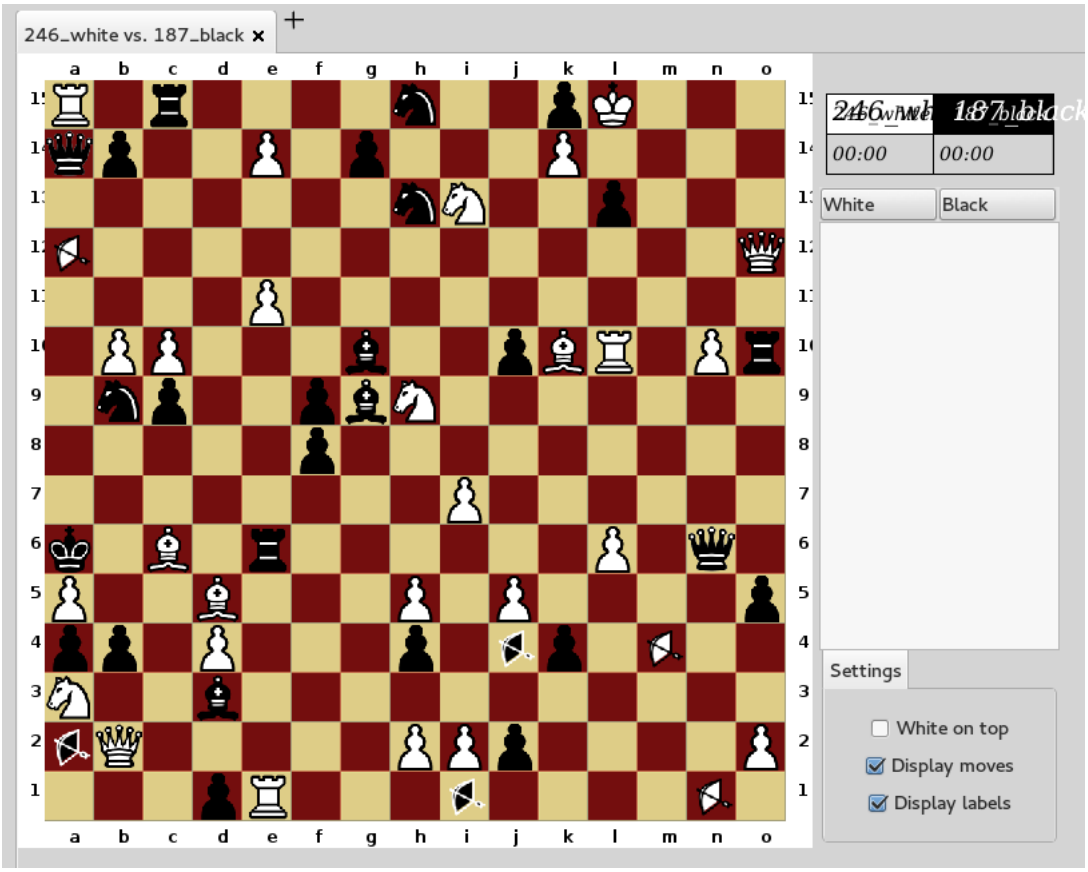
5 Variantes

Pour la variante des tailles, j'ai du reprendre la gestion de la taille de l'échequier un peu partout dans le code en modifiants les tailles hardcodées (généralement par les chiffres 7 et 8 (taille et taille-1)) en *chessboard.getSize()*.
La nouvelle pièce (Arrow) créée a été définie sur le même modèle que toutes les autres pièces (dans les packages *piece* et *behavior*). Cette nouvelle pièce peut se déplacer tout droit vers l'avant, et une case derrière ou sur les côtés.
Un pattern *Strategy* a été implémenté pour le choix du mode. 3 modes ont été développés :

- Le mode classique (où il faut une taille de 8, même si l'expérience utilisateur aurait pu être améliorée de ce côté)
- Le mode aléatoire (n'importe quelle taille), où les pièces sont placées aléatoirement.
- Le mode pions (n'importe quelle taille)), où la partie se réalise pion vs pion (et un roi). On peut obtenir d'autres pièces avec une promotion.

Chaque mode hérite d'une classe abstraite *jchess.core.setpieces.SetPieces* et implémente la méthode :

```
public abstract void setPieces4NewGame(Player plWhite, Player plBlack);
```



6 En visite

Dans le package *jchess.core.visitor*, on peut voir l'implémentation d'un design pattern *Visitor* là-aussi couplé avec un design pattern *Strategy* pour le choix du type de Visiteur. Ainsi chaque visiteur (ici *ScoringVisitor* et *TypeVisitor*) doit implémenter les méthodes :

```
void visit(Chessboard chessboard);
void visit(Square square);
```

qui seront ordonnées par les méthooes accept :

```
//Dans Chessboard.java
public void accept(ChessboardVisitor visitor) {
        for (int i = 0; i < settings.getSize(); i++) // create object for each
        // square
        {
                for (int y = 0; y < settings.getSize(); y++) {
                        squares[i][y].accept(visitor);
                }
        }
        visitor.visit(this);
}
```

```java
//Dans Square.java
public void accept(ChessboardVisitor visitor) {
        visitor.visit(this);
}
```

7 Décoration

Ici la classe Move a vu 2 nouveaux attributs, via le design pattern Decorator :

```java
private String duration = null;
private String comment = null;
```

Qu'on passe en paramètre via la méthode addMove :

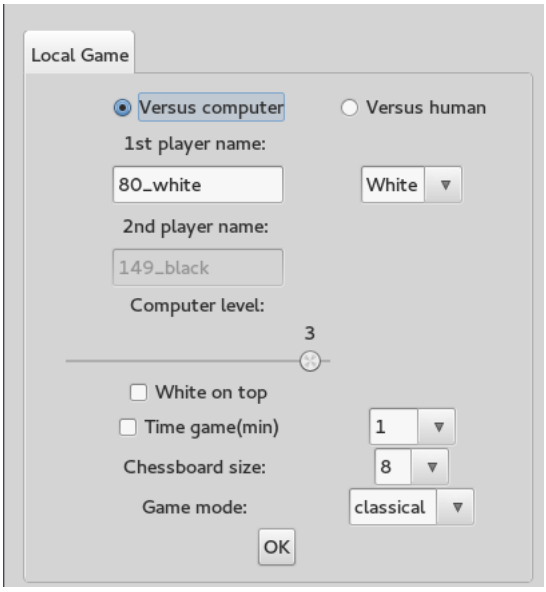
```java
public void addMove(Square begin, Square end, boolean registerInHistory,
Castling castlingMove, boolean wasEnPassant, Piece promotedPiece,
String duration, String comment);
```

8 AI

Ici encore, un design pattern de type *Strategy* a été mis en place pour implémenter les différentes intelligences. Ces classes sont visibles dans le package *jchess.core.computerai*. Ici chaque classe héritant doit implémenter la méthode :

```java
public abstract void move(Chessboard board);
```

De plus, les tests *testGlouton()* et *testAI()* on été implémenté pour cette partie.



Au final, 3 niveaux ont été implémentés :
- Le mode Random où les mouvements sont aléatoires.
- Le mode Glouton, qui mange les pièces de l'adversaire dès qu'il le peut

— Le mode MinMax, qui choisit le meilleur coup possible tout en nous rapportant le moins de points possible au tour suivant (joue avec un tour d'avance).

Note : dans tous les cas il est difficile de perdre contre un de ces modes.

9 Bilan

Le diagramme UML final du projet étant plutôt massif, il ne sera pas inclu directement dans ce rapport. Vous pouvez le trouver à l'url suivante : `https://github.com/AmarOk1412/ChessMDI/blob/master/uml/final.png?raw=true`.

Dans ce projet, j'ai mis en place quelques design pattern comme *Singleton* (Section 2), *Strategy* (Section 8), *Builder* (Section 4), *Visitor* (Section 6) et *Decorator* (Section 7).

Au final, beaucoup de choses n'étaient pas bien implémenté au début et des fonctionnalités plantaient (bug de déplacement, de promotion, de mise en échec, de logging, des tests manquants, etc). La partie la moins compréhensible était le lancement de l'application où on ne voyait pas comment il lançait l'application.

Enfin, l'intégralité du code est disponible sur `https://github.com/AmarOk1412/ChessMDI`.