# Projet MDI

*Jeux d'échecs & Design patterns*

*Frédéric Becker*
*Florent Guiotte*
*Yannick Jézéquel*

*ESIR 2 Informatique*

*2014-2015*

# Sommaire

Introduction ..... 3

Au commencement ..... 3

Diagramme de classe UML de l'application d'origine ..... 3

Identification des designs patterns déjà présents ..... 4

Design pattern d'interface ..... 4

Behavior ..... 4

Singleton ..... 4

Nettoyage ..... 5

Lancer une partie au chargement ..... 5

Décompte du temps ..... 5

Correction de bugs ..... 5

Variantes ..... 5

Conclusion ..... 5

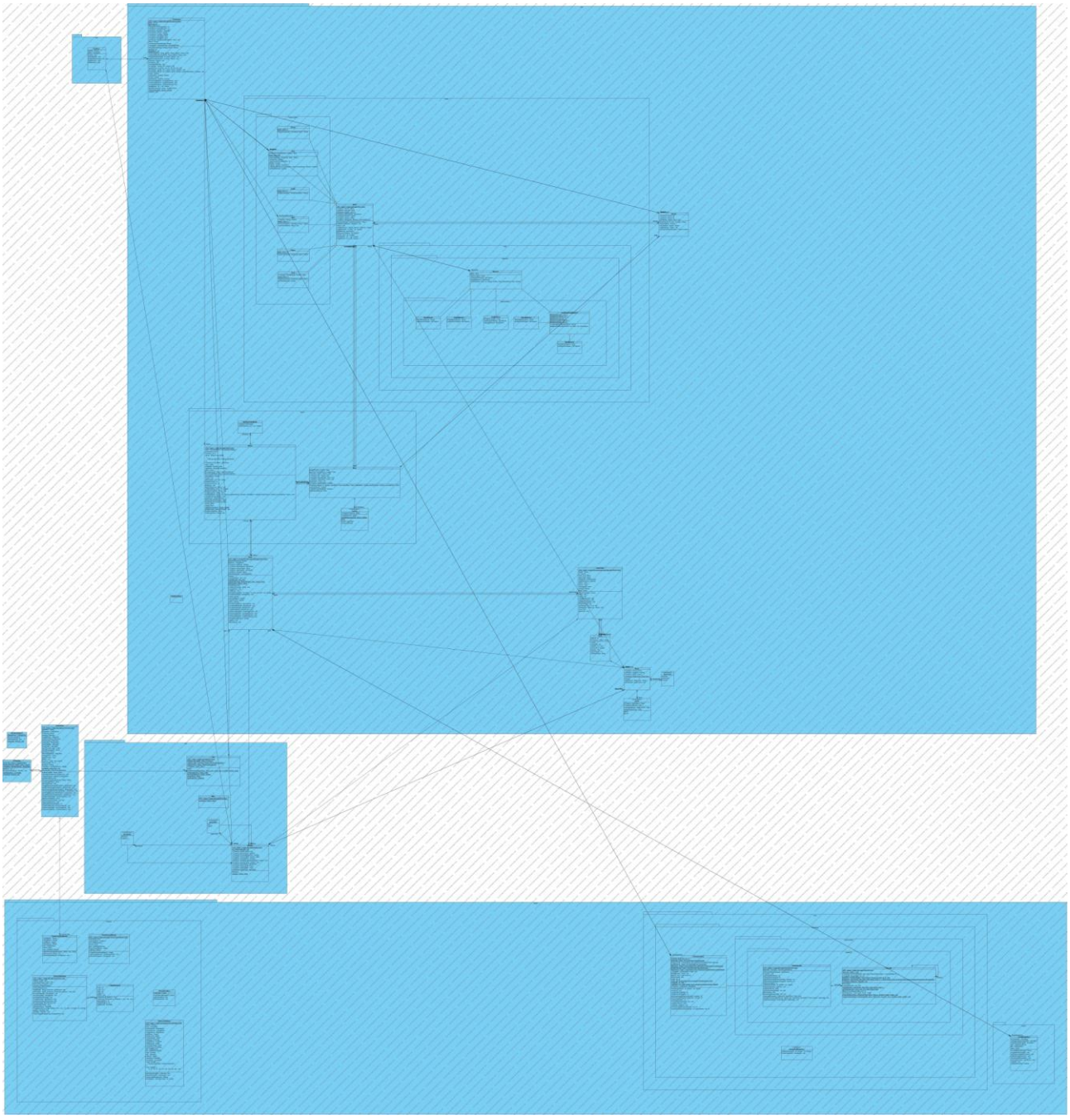
# Introduction

Le but de ce projet est de modifier un programme d'échecs déjà existant afin de le corriger et en réécrire une partie en s'inspirant des patrons de conception. Les patrons de conception (design pattern en anglais) sont un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

Le logiciel en question est « Java Open Chess » modifié pour l'occasion. Il présente de nombreux bugs et des comportements inattendus. Dans un premier temps, nous allons analyser le fonctionnement actuel du logiciel puis nous allons corriger les bugs critique et essayer de l'améliorer.

# Au commencement

## Diagramme de classe UML de l'application d'origine



## Identification des designs patterns déjà présents

### Design pattern d'interface

L'application se sépare en 3 groupes de classes distincts.

- Utils contient les données relatives à la partie, les réglages.
- Core contient le cœur du programme, la gestion de l'échiquier, des mouvements, de l'horloge.
- Display contient tout le code relatif à l'affichage.

Le design pattern observer est utilisé pour communiquer entre chacun de ces groupes, notamment pour affranchir l'affichage quand l'échiquier subit un changement. Cette organisation rappelle le patron de conception Modèle Vue Contrôleur qui propose de séparer le code en trois parties, dont la vue (ici Display), le contrôleur (Core) et le modèle qui contient toute les données de l'application (utils contient une partie des données seulement, mais se rapproche d'un modèle).

La mise à jour de l'interface se fait justement par observer, (la vue <<observe>> le modèle pour surveiller les mises à jour du modèle par le contrôleur).

### Behavior

Les pièces possèdent différents comportements. Ces comportements sont implémentés suivant le patron de conception template.

Le template method permet de définir précisément le comportement pour les pièces. En effet, ces pièces implémentent un comportement (Behavior) qui est abstrait. Les classes concrètes contenant les vrais déplacements (différents les uns des autres) étendent Behavior en héritant de cette classe. Ainsi quand chaque pièce cherche à connaître ses cases dans sa portée, la réponse donnée par la classe qui lui correspond.

Ce design est aussi utilisé pour donner des particularités à certaine pièces. Le roi possède en plus des déplacements et autres méthodes communes, des méthodes pour vérifier s'il est en lieu sûr etc...

### Singleton

Le singleton est un patron de conception qui permet d'assurer qu'une seule instance d'une classe pourra être instanciée. Ce pattern est utilisé dans ce projet par JChessApp et JChessView.

Ces deux classes ont en attribut statique une instance d'eux même. Lors de la construction de ces classes, l'attribut est vérifié, s'il est null, la classe est instanciée. On peut récupérer les instances en appelant n'importe quand 'getApplication()'.

## Nettoyage

### Lancer une partie au chargement

Afin de rendre l'expérience utilisateur plus agréable, nous avons fait en sorte de lancer une partie dès le lancement de l'application, Pour cela, il a fallu implémenter une méthode permettant d'obtenir un identifiant pour chaque joueur. Ensuite, une fois les identifiants obtenus, il suffit d'utiliser les méthodes habituelles de lancement de partie mais en remplaçant la méthode d'acquisition des noms des joueurs par les identifiants. Ces méthodes sont appelées dans le constructeur de JchessView ce qui permet bien de lancer la partie dès le début de l'application.

### Décompte du temps

Dans un souci de corriger le fait que le temps s'écoule avant le premier coup blanc, nous avons fait en sort d'enclencher le décompte uniquement lorsque le nombre de coups joués est égal à 1 et que l'initialisation du temps est différente de 0.

## Correction de bugs

Les 2 bugs que nous avons identifiés étaient : le déplacement du Fou et du Cavalier. Ces deux pièces manquaient d'un type de déplacement : le cavalier ne pouvait pas se déplacer en -1 ;-2, et le fou ne pouvait pas se déplacer vers le haut à droite.

D'autres bugs étaient présents mais plus compliqués à corriger. Comme par exemple le fait qu'un pion puisse mettre le roi en échec et mat ou le fait qu'un pion qui a traversé le plateau n'est pas promu car la fenêtre ne s'affiche jamais.

## Variantes

Rajout d'une classe Ninja dans le package Pieces.implementations, et d'une classe NinjaBehavior dans le package Pieces.Traits.behavior.implementation ,qui définissent ce qu'est un Ninja et comment il agit sur l'échiquier : même comportement qu'un fou mais limité à une case de portée. Ces deux classes ont été implémentées en suivant le design pattern déjà établi.

La variante de l'échiquier à taille variable ne nous semblait faisable car l'image n'aurait pas correspondu avec la taille réelle de l'échiquier. De plus la taille de l'échiquier est écrite « en dur » un peu partout dans le code ce qui complique encore la tâche.

## Conclusion

Grâce à ce mini-projet, nous avons pu identifier et mettre en œuvre différents design-pattern et nous avons pu découvrir la complexité du refactoring d'un code totalement inconnu et qui manque de flexibilité et de documentation. Certaines lignes de code étaient très obscures et leur sens nous échappe encore. C'est d'ailleurs cet aspect refactoring qui nous a le plus gêné dans la réalisation de ce mini-projet. Certaines fonctionnalités nous ont semblé bien trop difficiles à implémenter dans la situation actuelle et dans le temps imparti.