# Méthodes de Conception

**Jeu d'assemblage**


SAHIN Tolga
21801808

URAZOV Zhandos
21801012

SCHOEBELA Patryck
22012136

MIANGOUILA Meril
21810784


**Année universitaire 2020-2021**

# Table des matières

# 1 Introduction

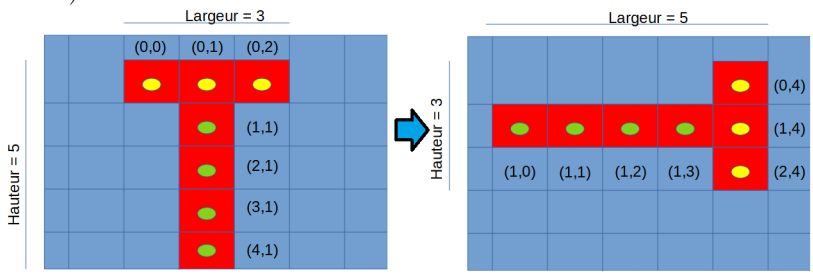
## 1.1 Logique de notre conception.

Un ensemble de cases pleines définissent nos pièces, avec des cordonnées définit depuis (0,0). Notre plateau est un ensemble de pièces. Le but est d'assembler chacune des pièces le plus proche l'une de l'autre.

# 2 Algorithmes

## 2.1 Rotation

L'algorithme d'une rotation horaire permet de modifier chacune des cordonnées des cases d'une pièce. On modifie x par y, et y par x - hauteur - 1. Puis, chaque rotation modifiant la dimension de la pièce. On adapte la hauteur par la valeur de la largeur, et la largeur par celle de la hauteur. Enfin, on incrémente un attribut orientation afin de pouvoir identifier l'orientation actuelle de la pièce (de 0 à 3).




Il en va de même pour la rotation anti-horaire. Sauf que l'on effectue la logique inverse. On modifie (y par x) et (x par la largeur - y - 1) et décremente l'attribut d'orientation.

## 2.2 IA BruteForce

Le but de cet algorithme est de déplacer les pièces triés (une par une tant que celle-ci peut bouger), d'une manière la plus optimale vers (0,0).

Afin de réaliser ceci, on récupère d'abord la somme des positions. Puis, on trie les pièces en fonction de la plus petite somme de positions. Ainsi, on a une liste trié de pièces les plus proches de (0,0). Donc, notre algorithme (solve) déplace latéralement vers (0,0) une pièce trié tant que celle-ci n'a aucune collision.

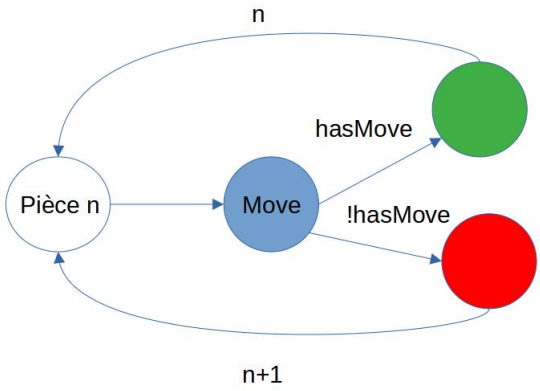



FIGURE 1 – Schéma de l'idée générale de l'algorithme.

## 2.3 Score

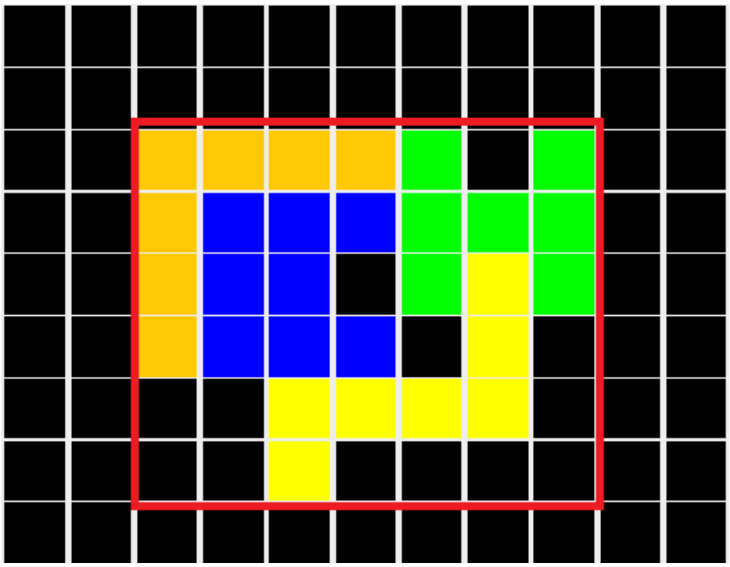


FIGURE 2 – Rectangle représentant le score.

Le but est de reproduire un rectangle comme sur l'image ci-dessus. Donc, notre algorithme de score récupère les cordonnées en (x,y) minimales et maximales dans le plateau. Puis, depuis ses cordonnées là on peut former 3 points du rectangle afin de calculer l'aire de celui-ci.

### 2.4 Parser JSON

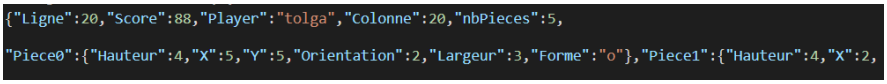



FIGURE 3 – Structure d'une sauvegarde.

Le parser JSON fonctionne grace à la librarire json-simple. Le but est de récuperer chaque clé écrite depuis notre sauvegarde afin de reformer un plateau et son ensemble de pièces.

En ce qui concerne les clés directement dans la 1ère "dimension", on les récupère tout simplement avec la clé. Cependant, pour les pièces qui sont des JSONObject. Tel que ci-dessous.

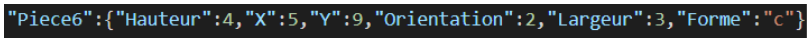



FIGURE 4 – Structure du JSONObject d'une pièce.

On va les récuperer en bouclant sur le nombre de pièces (vu que chaque clé d'un JSONObject de pièce est sous une forme "Piece" + i). Ainsi, on peut récuperer les clés à l'intérieur et former chacunes des pièces, puis les insérer dans un ensemble. Une fois que notre ensemble de pièces est complet, on peut créer et retourner le plateau final.

## 3 Patterns

### 3.1 Liste de patterns utilisés.

- Pattern MVC
- Pattern Observer
- Pattern Factory abstrait (3 fois -> PieceFactory, PieceRandomFactory, PlateauRandomFactory)
- Pattern Strategy (2 fois -> Pour l'IA et parser)
- Pattern Template Method (Subpackage pièce)

### 3.2 Résumé du modèle et des patterns.

Le modèle contient 5 sous package (piece, jeu, action, ia, parser) et 1 classe Game afin de générer une partie.

### 3.2.1 Subpackage piece

Le premier sous package est piece.

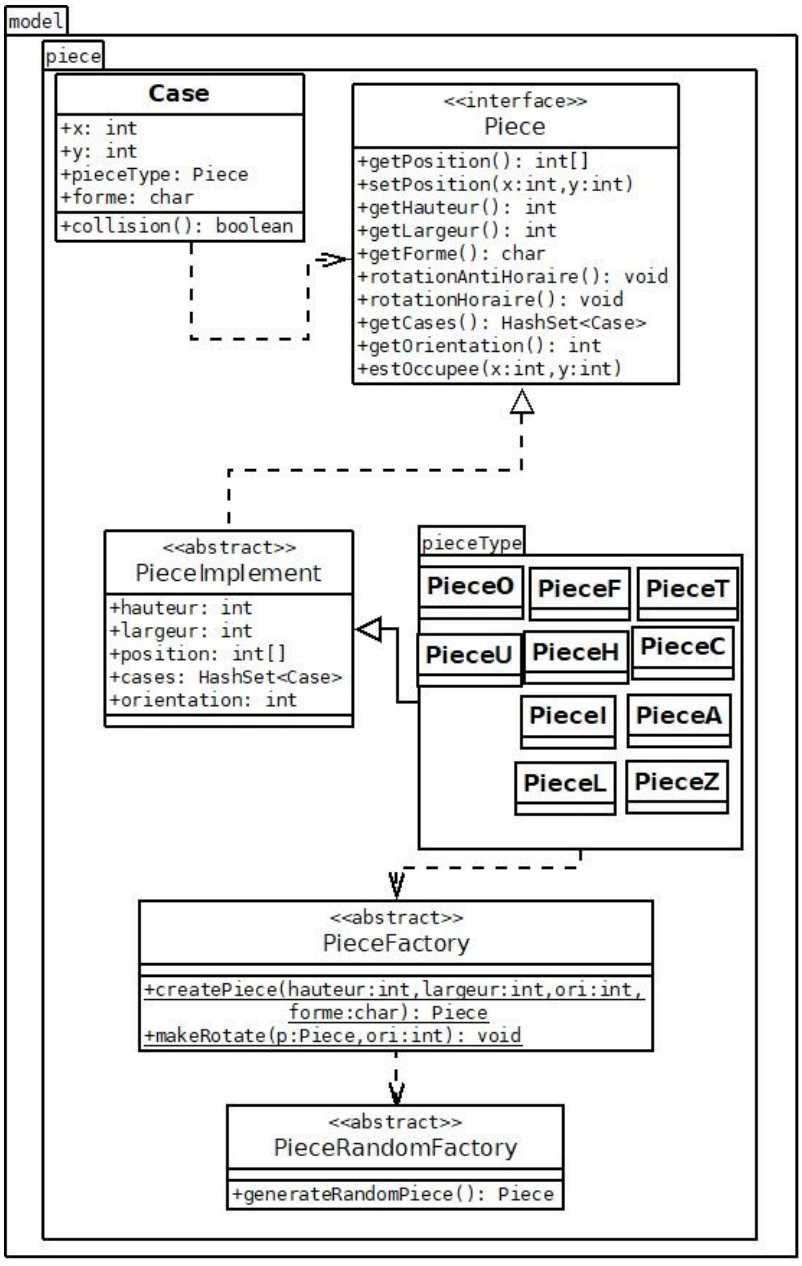



FIGURE 5 – UML sous package pièce.

Concrètement, on a construit nos pièces en fonction d'un ensemble de Cases pleines. Donc, une pièce dispose d'une dépendance vers la classe Case. Puis on réalise une implémentation sous la forme d'un Pattern Template Method, via une implémentation (PieceImplement) de l'interface d'une pièce. Puis, on en étend des sous classes afin de former différents types de pièces suivant la même structure.

Enfin, on a utiliser un Pattern Factory abstrait pour pouvoir créer une pièce en fonction de certains paramètres. On a aussi réaliser un Pattern Factory abstrait pour pouvoir construire une pièce aléatoire.

### 3.2.2 Subpackage action

Le sous package action.

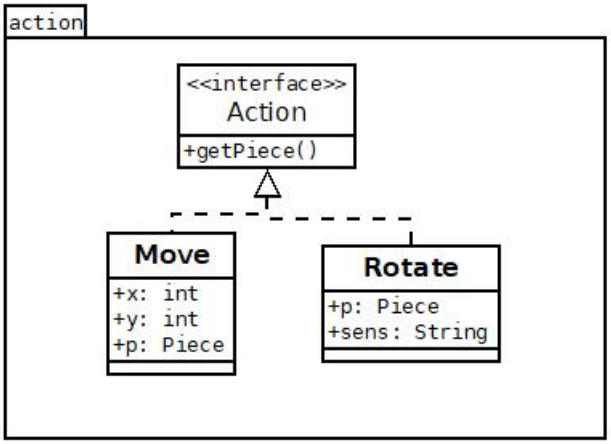



FIGURE 6 – UML sous package action.

Le second package est action. Son intéret est de définir une structure Objet pour chacune des actions d'une pièce dans un plateau. Donc, pour l'instant les seules actions nécessaires sont Rotate et Move.

### 3.2.3 Subpackage jeu

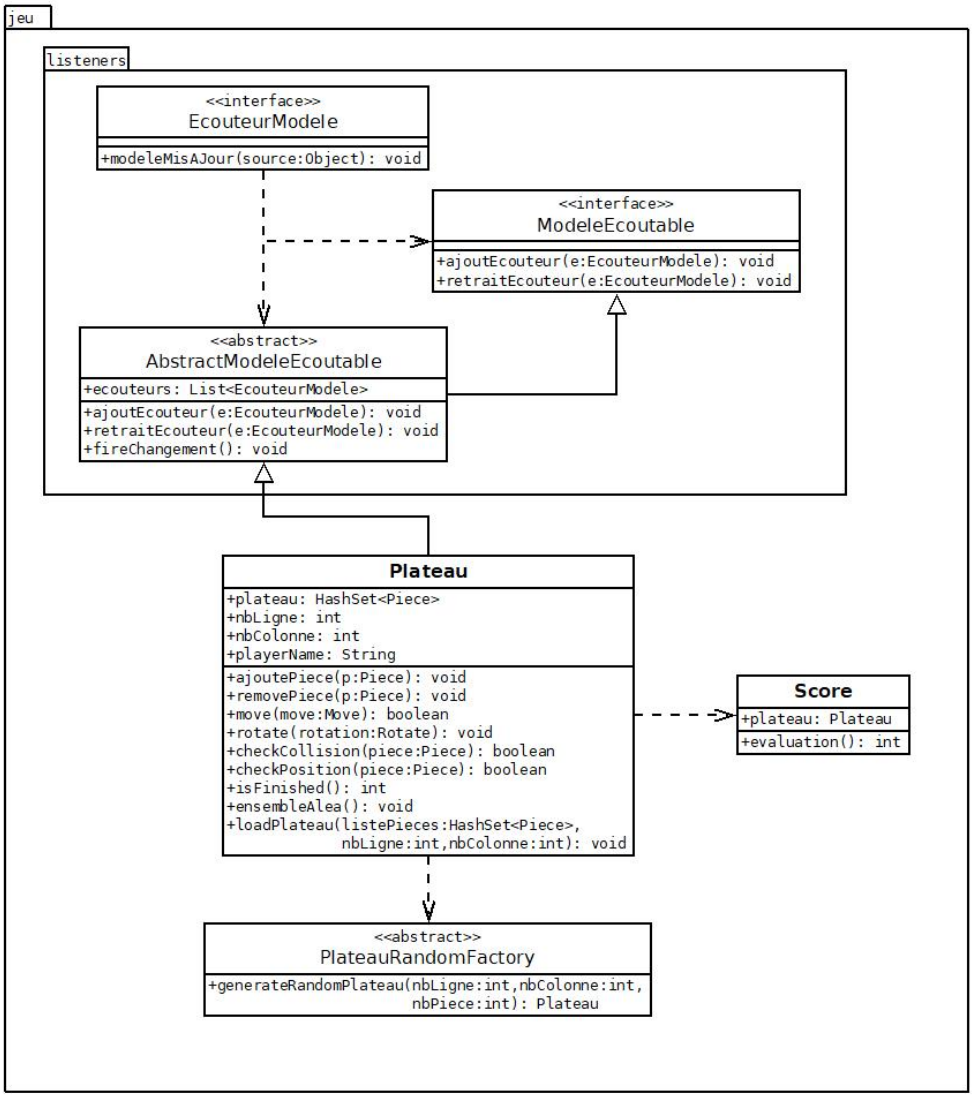
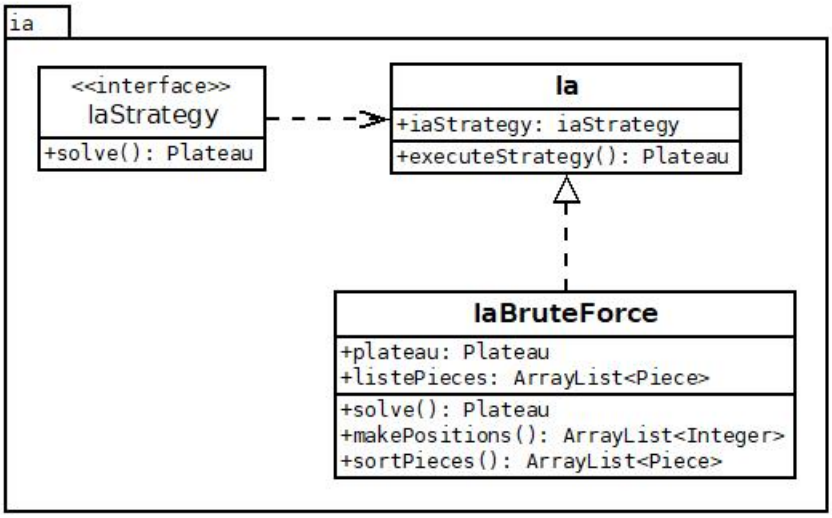



FIGURE 7 – UML sous package jeu.

Le sous package jeu contient le Pattern Observer. Notre classe essentielle est le Plateau, c'est là où on gère toute l'essence du jeu. On a effectuer une factory abstraite aléatoire du plateau afin de pouvoir générer un plateau aléatoire. Enfin, on a une classe Score pour pouvoir évaluer le score d'un plateau.

### 3.2.4 Subpackage ia

Le package IA est constitué d'un Pattern Strategy qui a pour but de pouvoir permettre l'implémentation de plusieurs stratégies d'IA. Actuellement la seule stratégie est l'IABruteForce. Afin de rendre ca plus performant on pourrait ajouter d'autres stratégies. Par exemple, une IA qui reprend la stratégie de l'algorithme KNN.




### 3.2.5 Subpackage parser

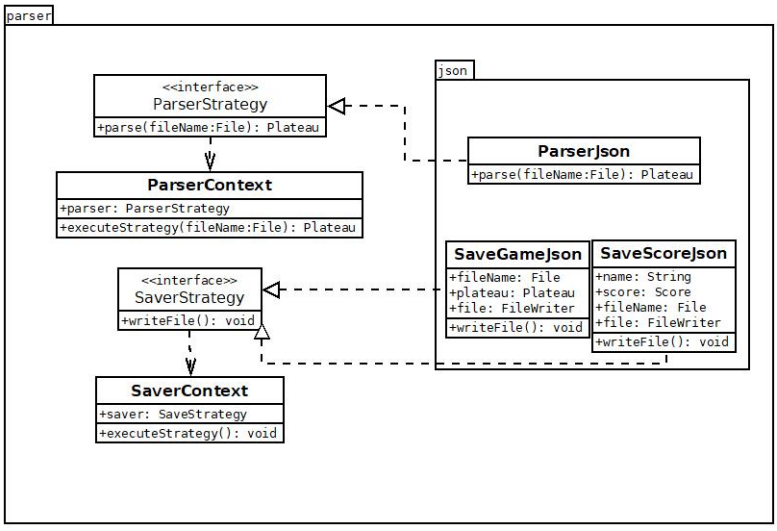



FIGURE 8 – UML sous package parser.

Le package parser a pour intéret de contenir les fichiers permettant la sauvegarde et le chargement d'un plateau. On a utiliser un Pattern Strategy, afin d'avoir plus d'ergonomie sur le choix des types de fichiers. Actuellement le seul format possible est le json. Mais, on pourrait ajouter de nouvelles stratégies et permettre de sauvegarder/charger sur de nouveaux types de fichiers par exemple en XML, en yaml ou en txt.

### 3.3 Résumé vue

On a représenter chaque cases par un panel. Puis, on a dessiner une grille de cases. Enfin, dans un panel au sud on a implémenter tout ce qui est nécessaire provenant du contrôleur.

### 3.4 Résumé contrôleur

Concrètement, on a construit tous les boutons nécessaires. Ainsi que le filtre de fichier JSON et les champs de nom et score pour les parties sauvegardés et chargés.

## 4 Conclusion

### 4.1 Améliorations souhaités/possibles.

On pourrait améliorer notre projet en enrichissant certains des patterns utilisés (notamment les Strategy). J'aurai bien voulu implémenter le pattern Chain of Responsibility aussi pour la détection des collisions. Enfin, côté vue, un pattern comme Decorator aurait pu être intéressant.