Rapport Projet

Objets et développement d'applications

PINEAU Sullivan, GIRARD Laurent

23/11/2015

Jeu d'Echecs

Depuis la racine du projet :

Compiler : make
Exécuter : ./bin/Echecs

Sommaire

Introduction

Ce projet a pour but de nous familiariser avec différents Design Patterns vus en cours. En effet, il nous était demandé d'implémenter trois Patterns différents au sein d'un sujet libre. Nous avons choisi de développer un jeu d'Echecs se rapprochant au mieux des règles officielles.
Avec notre programme, nous simulons une partie entre deux joueurs réels et chaque tour n'est pas chronométré. Les deux joueurs peuvent recommencer une nouvelle partie sans relancer le programme.

Les classes principales de ce projet sont les suivantes :

- Piece, classe abstraite dont on reparlera pour le Strategy.
- Player, un joueur se voit attribuer une couleur (soit Blanc, soit Noir), un nom et les 16 pièces avec lesquelles il va jouer (contenues dans un tableau). Ce joueur va pouvoir sélectionner une pièce.
- Chess, le jeu d'Echecs est composé d'un plateau et de deux Player. Le déroulement de la partie se fait au sein de cette classe. Chacun leur tour, les deux joueurs vont pouvoir déplacer leurs pièces sur le plateau après une sélection valide, et ce, jusqu'à la fin du jeu.

I - Présentation du jeu

a. Initialisation du plateau

Lors du lancement du programme, le nom des deux joueurs est demandé et sont ensuite créés. Chaque joueur possède 16 pièces (8 Pions, 2 Tours, 2 Cavaliers, 2 Fous, 1 Roi, 1 Reine) qui sont instanciées dès la création du joueur. Les pièces sont initialement positionnées de part et d'autre du plateau et possèdent leur propre mouvement.
Par défaut, c'est le joueur Blanc qui commence. C'est alors à son tour de jouer.

b. Le tour d'un joueur

Lorsqu'un joueur prend la main, sa seule action possible est de sélectionner l'une de ses pièces encore en vie. Ensuite, il peut soit annuler sa sélection soit choisir une destination pour sa pièce. Le déplacement est alors effectué sous certaines conditions :

- La case sur laquelle le joueur veut déplacer sa pièce correspond à un déplacement théorique de celle-ci (une tour ne pourra pas se déplacer sur une diagonale).
- La destination est soit une case vide, soit une case occupée par une pièce adverse.
- Il n'y a pas de collision sur le chemin entre la pièce et la destination choisie (sauf pour le Cavalier).
- Le déplacement n'entraîne pas la mise en échec de son propre roi.

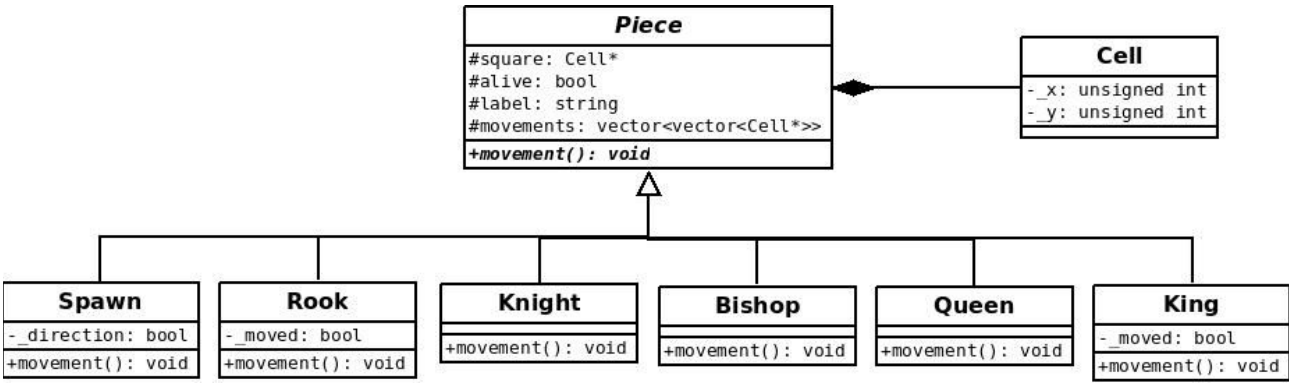
Une fois le déplacement validé et effectué, la pièce est déplacée sur le plateau et les conséquences de ce déplacement sont immédiatement misent en oeuvre (mise à jour du plateau et des coordonnées de la pièce, pièce adverse éliminée si nécessaire, etc.).
Si ce déplacement provoque la mise en échec du Roi adverse, son possesseur est alors obligé, à son prochain tour, de faire en sorte que son Roi ne le soit plus.
C'est ensuite à l'autre joueur d'engager un nouveau tour.

c. Fin d'une partie

La partie est terminée lorsque le Roi d'un des deux joueurs se retrouvent en position d'échec et mat. C'est à dire que lorsqu'un joueur à une possibilité de rencontrer le Roi adverse avec l'une de ses pièces dans un prochain déplacement mais que celui-ci ne peut pas se déplacer ou être protégé de sorte à ce que la mise en échec soit résolue. Le joueur à l'origine de cet échec et mat est alors le vainqueur.

II - Utilisations de Patterns

a. Strategy



Le diagramme ci-dessus représente le Pattern Strategy que nous avons intégré à notre projet.

La structure :

Tout d'abord, parlons de la classe abstraite Piece.
Celle-ci est composée de 4 attributs protected :

- square, la case sur laquelle la pièce se trouve
- alive, information si la pièce est en vie (=1) ou non (=0)
- label, lettre représentante de la pièce (Spawn → "S", King → "K", etc.)
- movements, stock les déplacements théoriques possibles de la pièce

Sur le diagramme, nous avons seulement fait apparaître la méthode en relation avec notre pattern : movement().
En effet, cette méthode permet de mettre à jour la liste des déplacements théoriques d'une pièce en fonction de la case sur laquelle elle se trouve (les collisions avec d'autres pièces sur le plateau sont gérées sur le plateau).

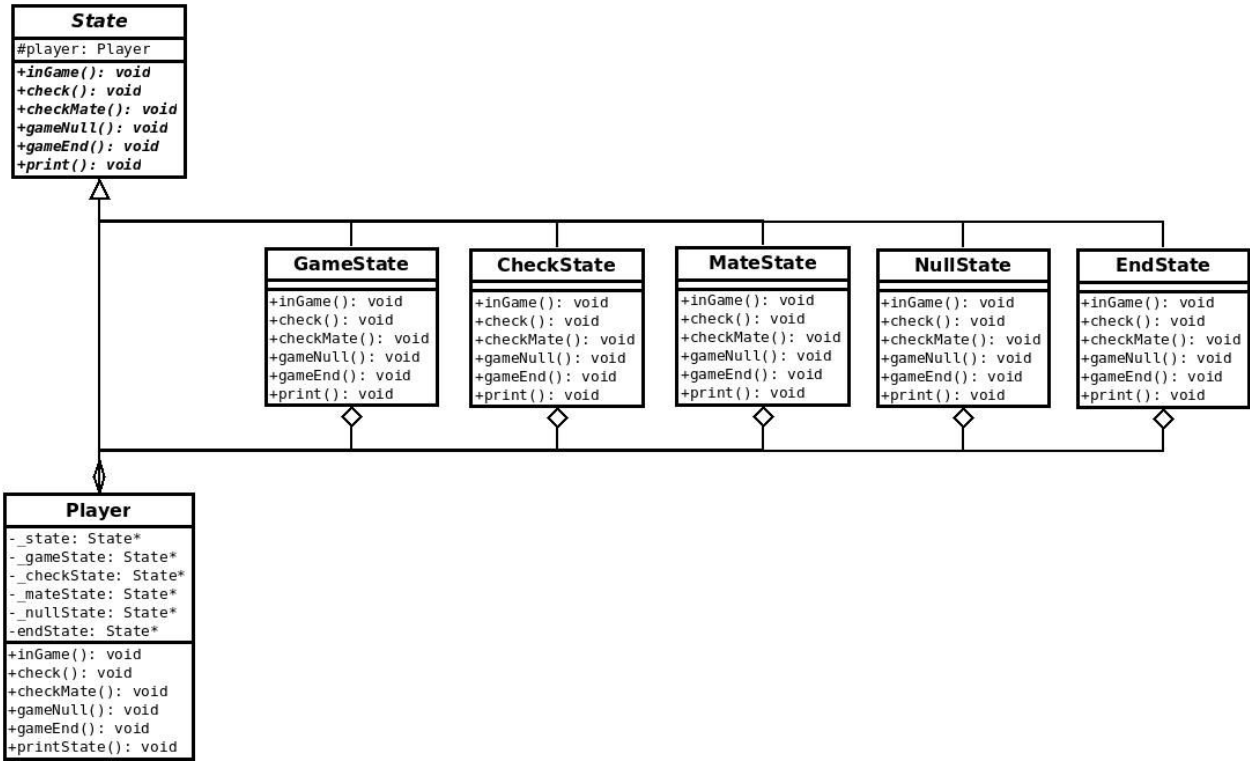
Parlons maintenant des classes concrètes (Spawn, Rook, Knight, Bishop, Queen, King).
Chaque classe concrète possède sa propre implémentation de la méthode "movement()". En effet, dans un jeu d'Echecs, chaque type de pièce à son déplacement propre.

L'intérêt :

Le Strategy nous permet ici d'appeler la méthode "movement()" lors de l'initialisation de la pièce ou après son déplacement et ainsi de garder à jour l'attribut "movements" quelque soit la pièce concrète.

Au sein de notre projet, nous utilisons une matrice de Piece* pour représenter le plateau de jeu.
Lorsque qu'un joueur déplace une pièce sur une case A(x,y) vers une case B(x',y'), il nous suffit d'appeler la méthode "movement()" sur la pièce sélectionnée après son déplacement. Ainsi, l'attribut "movements" contient tous les déplacements théoriques possibles sur le plateau depuis la case B.

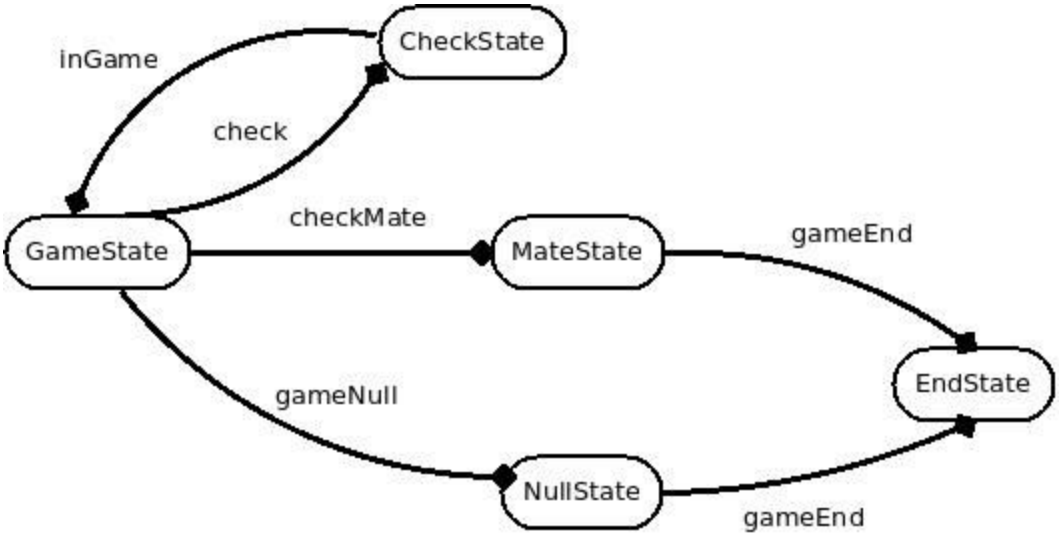
b. State



Ceci représente le Pattern State dans notre jeu d'Echecs.

La structure :

La classe abstraite State définit les transitions entre les états. Chaque état concret possède sa propre implémentation de ces transitions.
Les états sont les suivants : GameState, CheckState, MateState, NullState, EndState.
La classe client est ici le Joueur (=Player) avec pour attributs les différents états possibles ainsi qu'un état courant qui "naviguera" entre les différents états.
Voici le diagramme d'états correspondant :

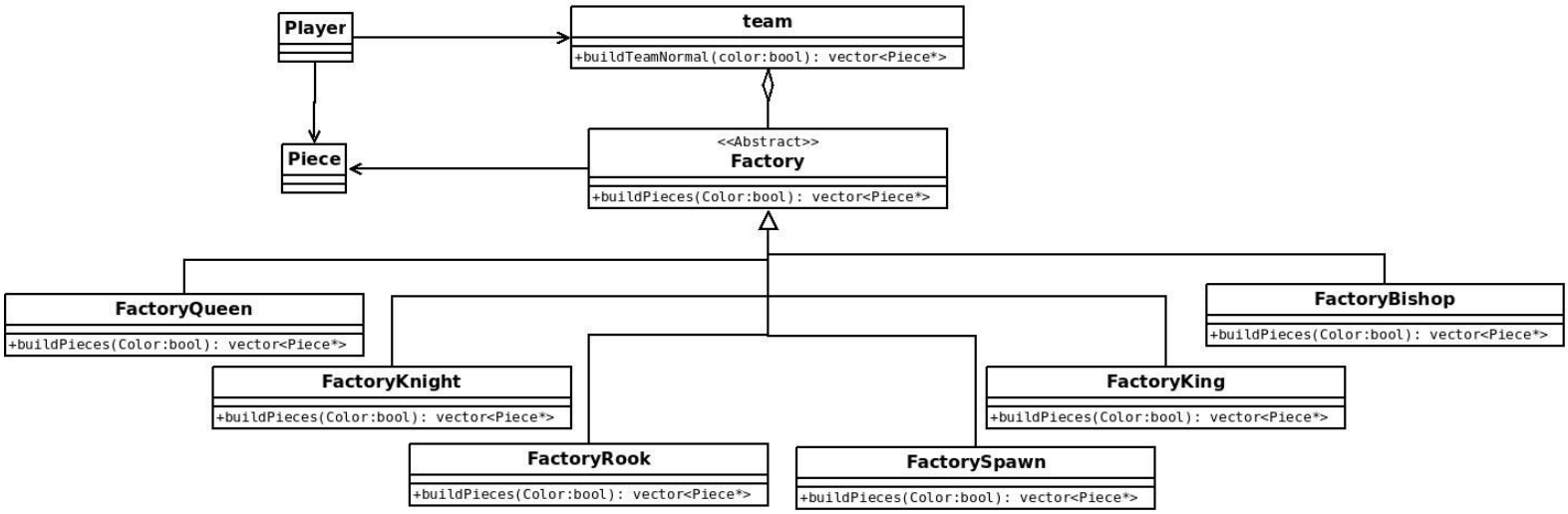


Sur ce diagramme, l'état initial correspond à l'état GameState et l'état final à EndState. Il permet notamment de distinguer les deux états qui pourraient traduire une fin de partie.

L'intérêt :

L'état courant se mettra à jour automatiquement en fonction des situations et lorsque l'état courant du joueur sera modifié, il y aura un affichage de cet état sur la sortie standard.

c. Factory Method



La structure :

La classe principale de ce pattern permet de construire une liste de pièces initialisée pour une partie dite "normal".
Ces pièces sont initialisées au bon endroit en fonction du booléen (false = White).

L'intérêt :

Lorsqu'un joueur est créé, les pièces qu'il possède sont initialisées en fonction de la nature de chaque pièce et aussi en fonction de sa couleur.
En effet, chaque pièce sera positionnée différemment : les Rooks seront respectivement en coordonnées (0,0) et (0,7) pour le joueur Blanc.

III - Améliorations possibles

Quelques éléments d'amélioration possible :

- Afin d'implémenter la fonction de "retour en arrière" après le déplacement d'une pièce (ou "undo" en anglais), il nous aurait été possible de remplacer notre pattern Strategy par un pattern Command.

- Notre version actuelle n'est jouable que par deux joueurs (différents si possible…). Une amélioration envisageable serait de proposer un tutoriel à un joueur solitaire ou encore de lui proposer de jouer contre une Intelligence Artificielle.

- Proposer un rendu graphique plus agréable que la sortie standard. Ce qui pourrait entraîner la mise en place d'un MVC et ainsi séparer la Vue du Controller (notamment à l'aide d'un pattern Observer) pour rendre le code plus lisible. De plus, il serait bien plus pratique que le joueur puisse sélectionner directement à la souris (ou à la main pour une version tactile) la pièce qu'il souhaite déplacer plutôt que de rentrer les coordonnées de la pièce ou de la destination manuellement.

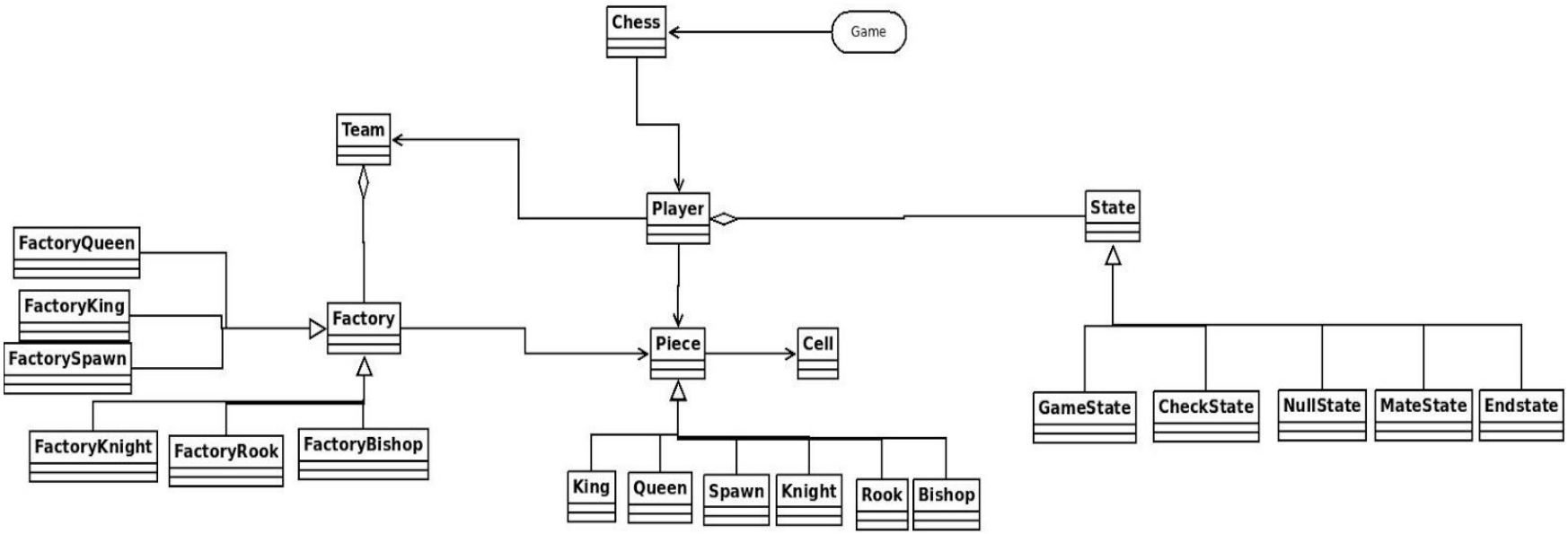
Conclusion

Ce projet, en plus de nous avoir familiarisé avec l'utilisation de quelques Patterns au sein d'un programme, nous a également permis de nous confronter à certaines difficultés concernant la conception d'un projet depuis une simple idée. En effet, nous avons consacré de longs moments à sa structure, après plusieurs schémas différents, nous nous sommes arrêtés sur celui fourni en dernière page.

L'approche "bottom-up" fut très utile pour implémenter le cas des différentes pièces. En effet, il serait facile d'ajouter une nouvelle pièce au jeu, si l'envie de créer un Dragon vient, il suffirait de créer une classe Dragon, nouvelle classe fille de Piece (cependant, la modification de Piece entraîne une modification de chaque classe fille).
De même qu'il serait facile d'ajouter une nouvelle FactoryConcrète

Malheureusement, la version rendue du projet ne sera pas finie dans le sens où un certain nombre de règles n'ont pas été réalisées (la "prise en passant" et certains cas de partie nulle).

Il aurait également été intéressant de se pencher sur le chronométrage des tours de chaque joueur et ainsi proposer un programme pouvant simuler une partie compétitive.


Chess
Game
Team
Player
State
FactoryQueen
FactoryKing
FactorySpawn
Factory
Piece
Cell
FactoryKnight
FactoryRook
FactoryBishop
King
Queen
Spawn
Knight
Rook
Bishop
GameState
CheckState
NullState
MateState
Endstate