LP24 REPORT
CREATE A THREE-DIMENSIONAL CHESS GAME

MAXIME BOURGEOIS & NATHAN OLFF

JUNE 2013

Table des matières

I – Introduction

Description of the subject.

The classical chess game is one of the world's most popular games, played by millions of people worldwide at home, in clubs, online, by correspondence, and in tournaments.

Throughout the centuries, many variances of the chess game have been invented: 3-player chess, Hexagonal chess, Rhombic Chess, etc…

One of these variant was created by Gene Roddenberry in the 1966 science-fiction show Star Trek. It is a three-dimensional variant, allowing pieces to move in three physical dimensions.
Four boards (two for each player) are movable and can also move in three dimensions.

In popular culture, this chess variant has been mentioned in *Doctor Who*, *The Big Bang Theory* or even in Audi commercial.

We had to create an 3D-chess game with Java.

- The game have to be played easily and by everyone,
- The game is created in 3D but can be displayed in 2D,
- 2 small boards per player have to be movable with some contraints (see rules below),

- Bonus : an artificial intelligence can be made.

Describe objectives & Problem statements.

The objective of this project was :

- Improve our skills in Java,
- Learn how to create a small game with Java,
- Learn how to create a GUI and how to link it with the rest of the program,

- Optional : learn how to create a random AI

Rules

The first problem we encountered when we began was the fact that there are no real rules defined within the Star Trek franchise. While searching on the web, we found many different versions of the rules. Their basic principles were the same, but some details differed like the initial position of the pieces, the possible movements for the movable grids. Below is a transcription of the rules we implemented in our game.

Boards :

- There are 3 main boards (4×4 squares), and 4 attack boards.
- The attack boards can move on the top of any corner of a main board.
- The main board are located on levels 2, 4 and 6.
- The attack boards can go on level 3, 5 and 7.
- The first main board is called the White Boards, the second is the Neutral Board and the last one (on level 6) is the Black Board.
- Each main board has 2 common rows with the board below.
- Each attack board is linked to a corner. Left corners of the main boards are called Queen Levels (QL), and right corners are King Levels (KL). They are named like this because both Kings and both Queens are respectively on the same side of the boards.
- Therefore, any attack board as a common coordinates (on X and Y) with the main board on which it is attached.
- At the beginning, 2 attack boards are located on the extreme corners of each Black and White Board. That gives for each attack board an original owner, depending on its original position.
- An attack board can be moved to an adjacent corner only if :
    - There is no piece on the board. Only the original owner of board can then move it.
    - There is only one piece on the board. Only the owner of the pawn can then move the board.

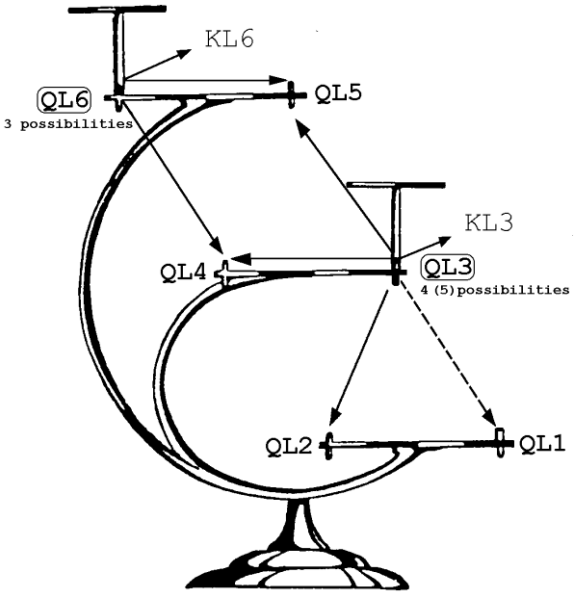


Figure 1 - accessible corners

Piece movements :

- Any piece moves like on a normal chess game, except it can go on any board, at any level.
- A piece, on any individual square, blocks the ability of other pieces to move at all levels

further than its location. But the moving piece may land above or below the occupied square and continue its move on the next turn.

- A piece cannot go straight up of straight down without moving horizontally.
- If a pawn reaches the furthest row of the chess game (on a main board or an attack board), it is promoted to another rank, which can be Queen, Rook, Bishop or Knight.

The game is over when a player's king is captured by the enemy, or when a player has only his king left alive.

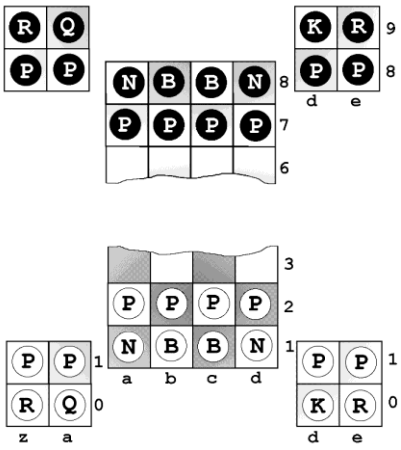


Figure 2- original positions of the pieces

II – Project

MVC pattern

We used the MVC design pattern as possible as we can in order to better decompose the code as

Nathan worked on the model and Maxime worked on the view and the controller (package *chessGUI*).

We tried to stick to the classical MVC pattern, but had some difficulties at some point respecting it. It allowed us to easily start a new game, save it, load it later. As the view is completely made from the model, we could change the interface quickly and easily without changing the model, we can pause a game and restart it later (save/load functions) and so on …

This design pattern was really a great addition to our project.

Model

The model is the part of the program that implements the game engine. It is completely independent from the view and the controller. It has 3 major components: the *Board* (which is in fact a collection of grids), the set of all pieces, and the list of players. For each movement of piece or board, the model calls a method that checks if the movement is possible for the current player (and consistent with the rules). Everything within the model is located by 3d-coordinates.

The Board

In the model, the Board is a collection of 7 *Grids*, 3 fixed and 4 movable. Each *Grid* has 2 coordinates: the first case and the last case contained on the *Grid*. From there, we can compute the size of the grid and its type. Movable grids are also linked to the *Corner* (i.e. the rules above) of a fixed grid on which it stands.

Pieces

Each piece has a coordinate (class *Coord*), a *Color* (Black or White), and a list of relative movements. Anytime we want to move a piece, a method of the model computes a list of accessible squares using the relative movements. To manage the different types of piece, we have different classes that inherit from the abstract class *Piece*.

Players

There are two types of player : *HumanPlayer* and *BotPlayer*. In a perfect MVC program applying the abstraction principle, both would call the same methods and work the same way from the outside. As we tried to do it like this, we were put face to face with the notion of threads. Apparently, threads would have helped us synchronising the two types of player during a game. Finally, we created a special method *play* in the *BotPlayer* that chooses randomly a piece or a grid to move.

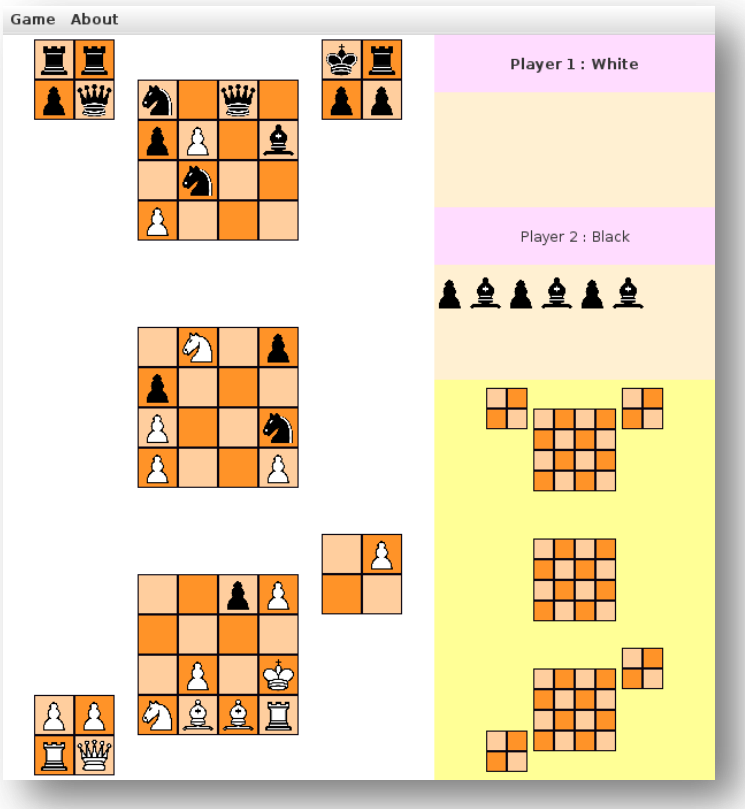


Figure 3 - Screenshot

GUI

Main chessboard

To create the main chessboard, I used a *GridBagLayout* that I filled with cases.

A case is a *JPanelImage* which is a *JPanel* containing an image. A case exists or not, if she does not exist her background colour is the chessboard's background colour.

There are two types of coordinates for a case : a *Coord* which is the original 3D coordinates of the model and a *CoordGraph* which corresponds to a translation of the 3D coordinates on the 2D graphical window.

A map stores the cases with the case's *CoordGraph* (2D coord) as key and the case as value. Therefore, we can access to a case by the way of the case's coordinates, 2D or 3D.

The aim of the main chessboard is to move the pieces.

Small chessboard

I created the small chessboard the same way I created the main chessboard.
Small cases have the same *Coord/CoordGraph* than main cases so every case has a small case with the same coordinate.

A map stores the small cases with the small case's coordGraph as key and the small case as value. We also access to a small case with the map.

The goal of the small chessboard is to move the movable boards.

Information panel

The purpose of this panel is to display two kinds of information:

- Information about the player (name, color). The current player is displayed in bold font.
- The player's dead pieces

The implementation for changing a player's name is possible in the model but has not been implemented in the graphical interface.

Each time a player loses a piece, the killed piece is added on this panel such that a player can know at any moment which pieces he lost.

Listeners

MouseListener

The *MouseListener* is our controller. It mostly manages mouse clicks and allows the user to move his pieces/boards. It also manages the promotion of pawn (when a pawn arrives at the other end of the chessboard; see the rules for details).

It recovers the clicks of the user and checks the movements with the model. If the movement asked is correct, the controller calls the refresh methods of the view that apply the changes on the *Model* and tell the *View* to refresh the screen.

PlayerListener (interface)

After a human or a bot has played (moved a board or a piece), the *fireHasPlayed* method is called. This method changes the current player in the model.

Artificial Intelligence

We created a random A.I. which can move pieces and attack boards.
A bot attacks an enemy piece if it is possible.

The choice between human and bot is made at the beginning of the game with the display of a popup window.

Save/Load functions

As we were developing our game, we encountered difficulties about the debugging of the end of the game and the movement of attack boards because we had to play a few minutes every time in order to test the program. This is where we had the idea of saving the game and loading it later, in order to save time.

The model uses the *JDom2* API to save the components of the model into a XML file and decode this XML file to load a previous game. These functions allow the player to pause to game and continue it later.

III – Conclusion

With this project, we learnt how to create a good software by applying the MVC pattern (at least we tried).

We also learnt how to create the GUI with all the problems associated to it: layout, interactions with model/controller ... The project which was not really easy for our level in Java gave us good notions in Java/Swing.

We planned to put *JPanel* representing boards for the small chessboard instead of small cases and to write their level on them. We did not change it because the controller was already using the small chessboard (with all the small cases) and as we implemented it a long time ago, changing it was a bit hard and gave us a lot of bugs. We also planned to display a side view of the board in order to improve the game experience.

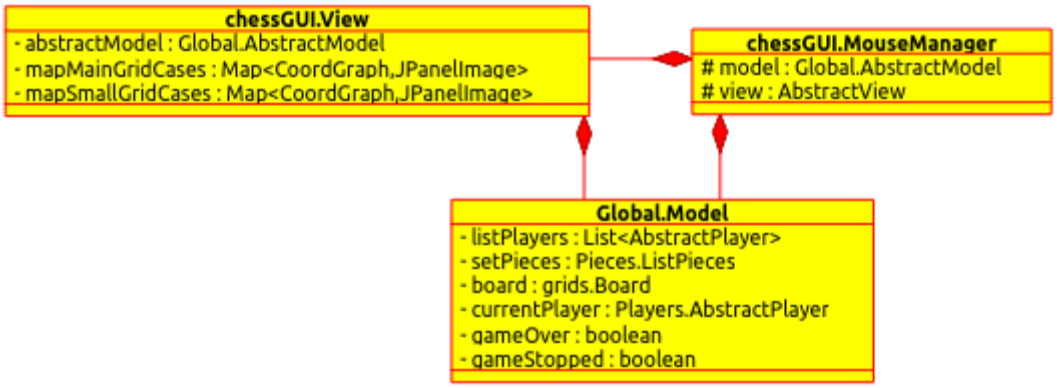
We also thought about storing all the moves done in a stack. With this, we could have replay a game, show the last moves done or even cancel a bad move.

The A.I. is still basic and a lot of improvements can be done to it. For example, different levels of difficulties, add probabilities

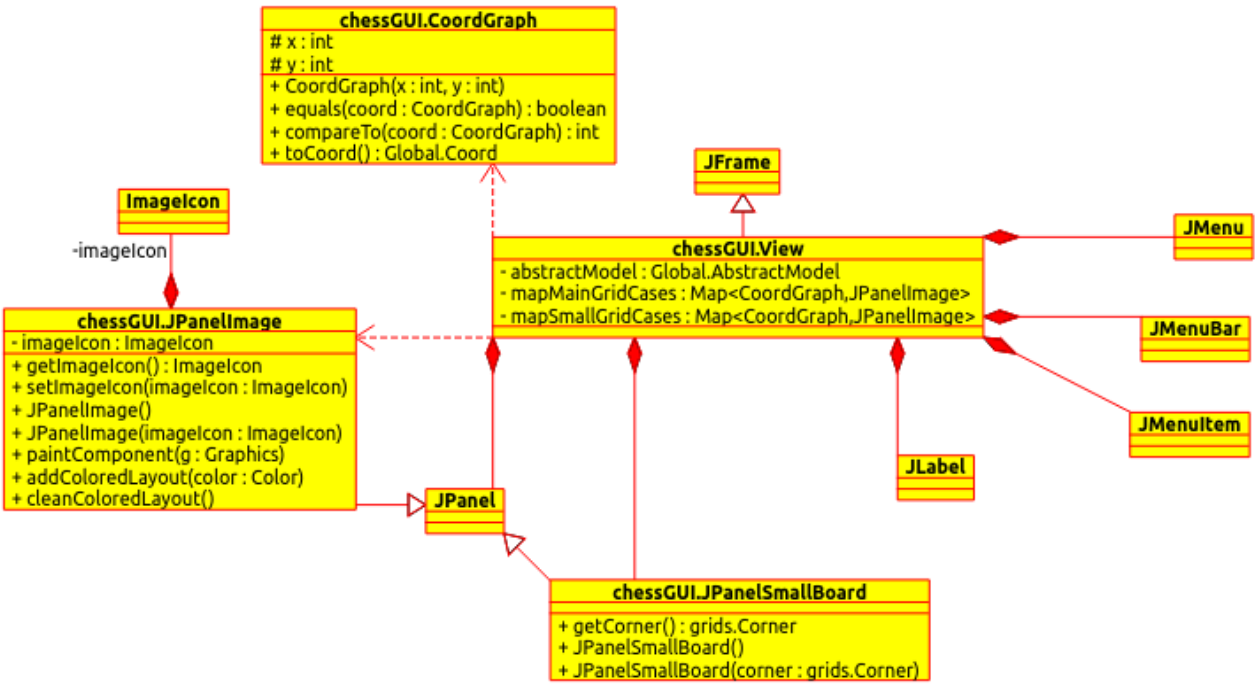
We could have used threads/sockets but it appeared to be too complex without lesson about it and hard to integrate to the project.

To conclude, we learnt a lot with this project but a lot of improvements can be done and we still have concepts to learn in Java.

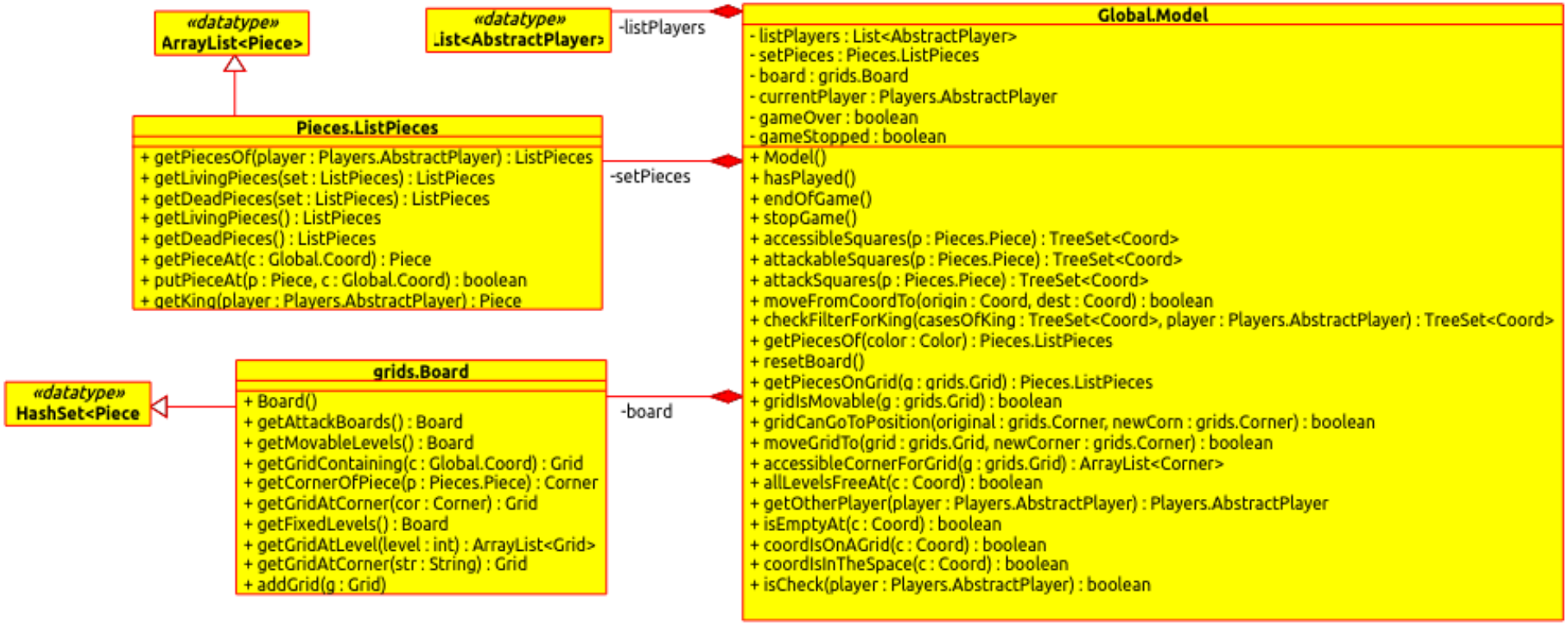
Annexe : UML Diagramm



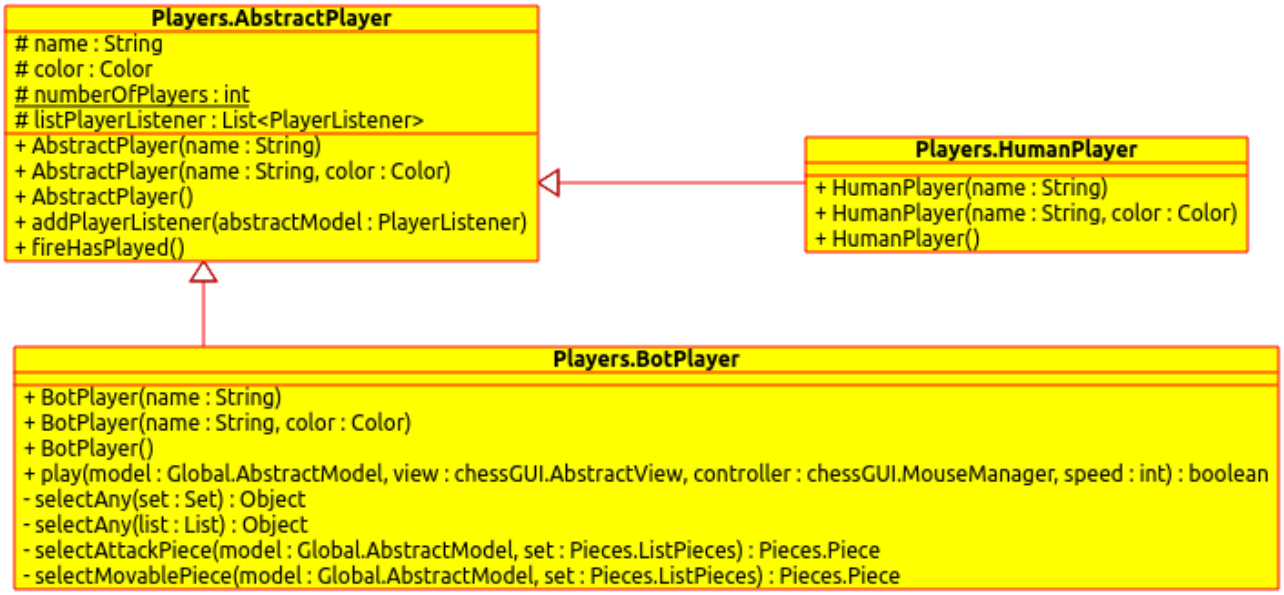
Annexe 1 – MVC pattern



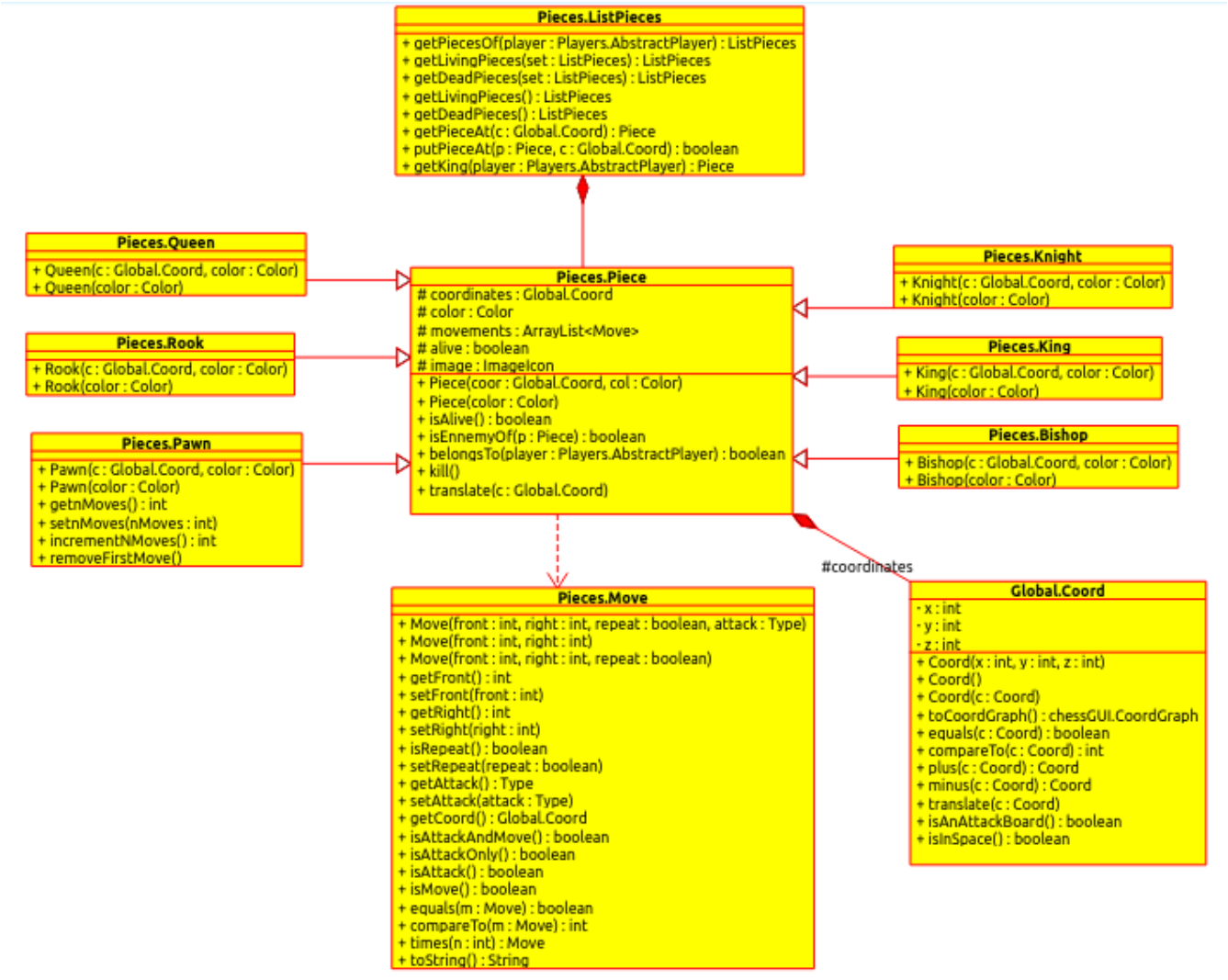
Annexe 2 - View



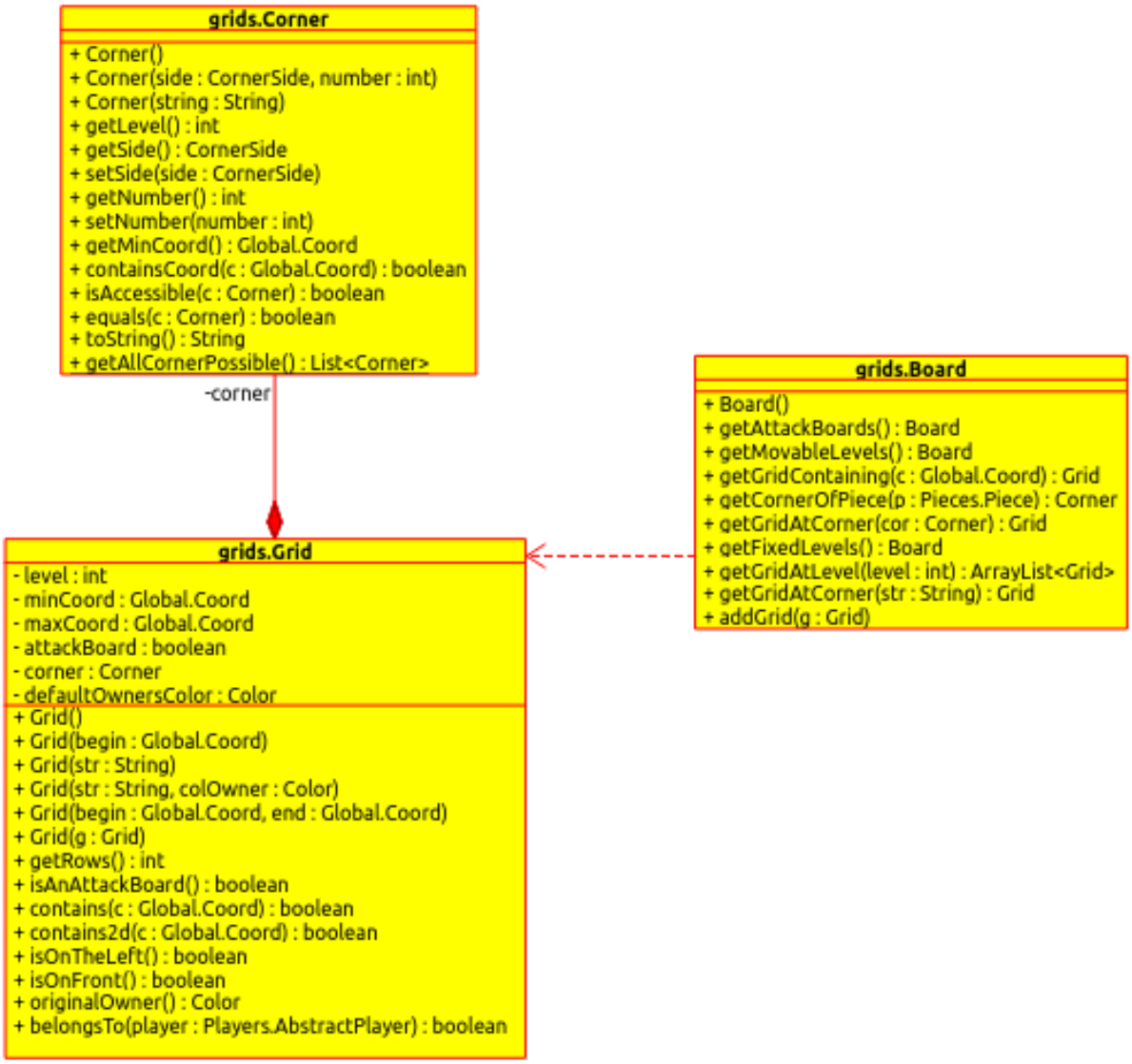
Annexe 3 - Model



Annexe 4 - Players



Annexe 5 - Piece



Annexe 6 - Grid