JavaChess

Developed by Jevgenij Semionov

1. Problem analysis

The goal of this project is to create a highly portable, fast and user-friendly PC chess game. In particular the program:

- Implements all standart chess rules
- Permits player versus player matches on single station
- implements a highly scalable structure

The chess game is played entirely in 2d. The chess pieces are moved by players using the mouse by dragging and dropping them on desired squares. When the user clicks on the piece that he desires to move, the squares that are legal moves for the piece get highlighted.

The main problem of this project that i found as i coded it, that in chess, no rule holds in general. So for instance, you can say that every piece eats as it moves, EXCEPT the pawn. Or that you can only move one piece at a time, except when you perform castling. So these little exceptions, make it hard for the chessgame to fit in object oriented architecture in a natural and scalable way, or atleast i found it so.

What really brings a breath of a fresh air to the user expirience is that the pieces are oriented vertically and non horizontally as usual. It doesn't change the gameplay or any particular rule, but it shows you the chessboard from a perspective you're not used to, so the player is enhanced to see some new solutions.

2. Project architecture

The application implements model view contoller pattern, in particular we have the model, that is the visible chessboard that interacts with the user, but doesn't implement any chess logic, the logical chessboard, that implements the logical chessboard and all the chess rules and the controller that maps all the user input to the logical chessboard and in rare cases it's used by logical chessboard to signal something to the visual one. I also found it natural to incapsulate every single chess piece as a class on it's own, and altough it can be discussable, no chess piece contains information about it's position on the board. I find it more indipendent and incapsulated this way, altough it resulted in a more difficult implementation that i think would pay off in the eventual future developement.

Classes

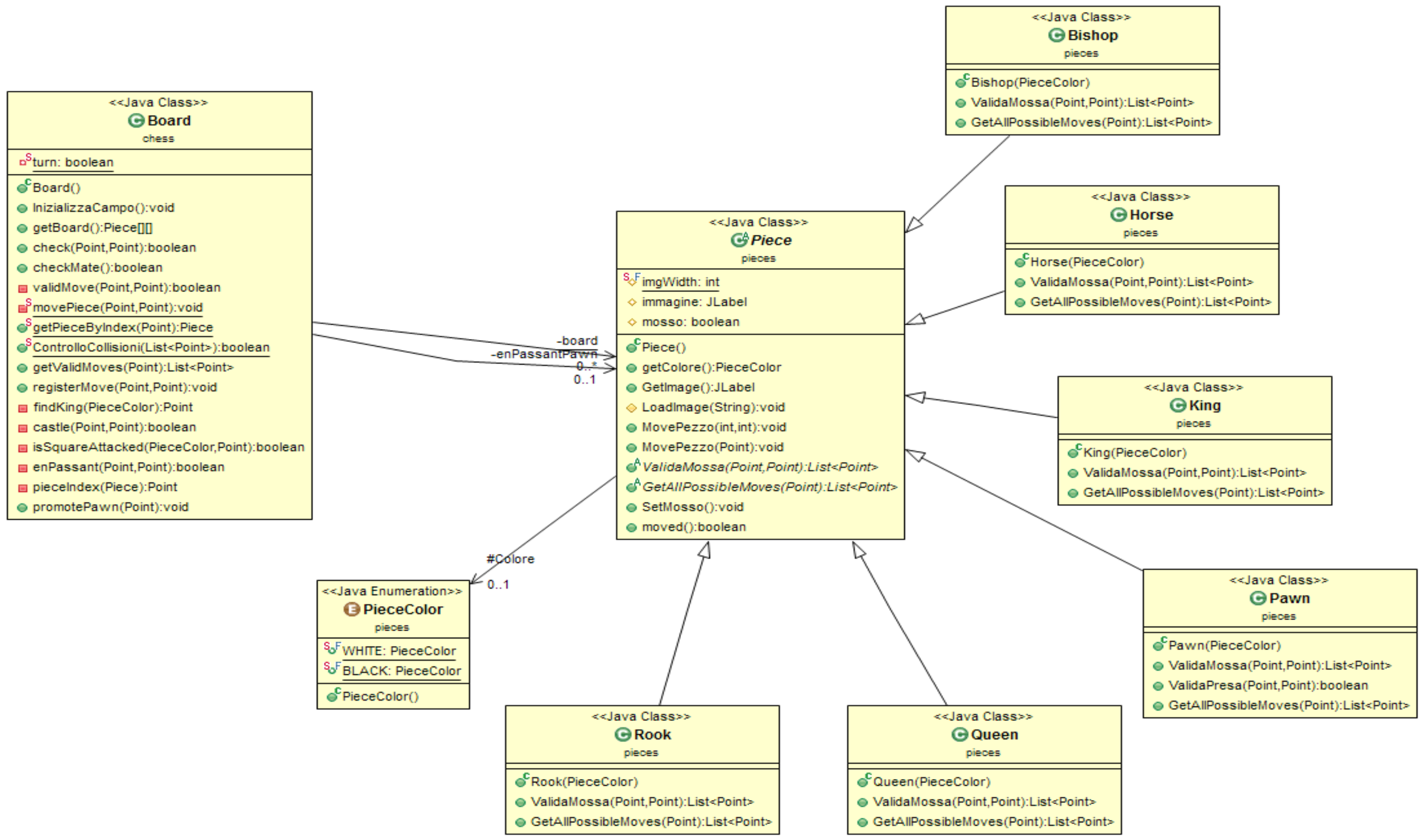
Pieces (entire package)

So let's start from the pieces. We have piece package which contains the abstract class Piece, all the single piece types classes and the enumerator PieceColor which i use in every piece to hold it's color. Every chess piece class inheritates from an abstract class Piece which declares some abstract methods, which should be implemented individually for every piece, methods like getAllPossibleMoves which returns a list of all possible moves for the piece, and we all know that every type of chess piece moves differently, so the rules must be implemented individually. It also implements some methods like loadImage which loads the chess figure image. Infact that may be a mistake to keep the image of the figure in the concrete piece class, because the piece is a logical (model) entity and the image is something a view entity, but i find it so natural to incapsulate everything that goes in general about a piece withing it's own class that i just go for the risk and leave it this way. Also as you can see every piece class initializes it's image from a hardcoded string which is not good. I should fix that, infact i'm planning to implement a skin system. Further, we have all the individual piece types that inheritate from the abstract Piece class, and they are basicly all the same except the Pawn class which need an extra method validTake that is called by the Board class (which is the logical chessboard implementation class) to verify if the pawn can perform the take. This is a little bit messy, that is due to the fact that pawn takes differently from how it moves. I could implement the take logic of the pawn inside the Board, but as i said, i wanted to incapuslate all the rules that are direct to a particular piece, inside that pieces class.

Board

Now let's take a look at the Board class. This is the main model class in my architecture. It hold the 8X8 Piece matrix (board) which rappresents the logical chessboard, which is the core of the game. The board also has other two members which are boolean turn which i use to keep track of current turn, and the Piece enPassantPawn which i use to store the pawn that has **just** performed the double opening. So basicly when the Controller creates the instance of a Board class, it initializes board with all the pieces and waits for the Controller to call one of its public methods which must corrispond correctly to the user input. It also has some private methods like validMove which is responsable for validating if a concrete move is legal or not. The lack of my implementation is that to use the board, one needs to know the entire logic behind the pieces and the board. This is due to the fact that the pieces know nothing about the board, so for instance when we ask the concrete piece if it can perform a particular move, one could expect that it's answear is deffinitive, but it's not. The piece only tell's us if the move respects pieces type, and doesn't look at all at the current board situation. So to check if the move is legal, one should also perform collision detection, check condition ecc which is maybe not that intuitive.

That's the model scheme



<<Java Class>>
Board
chess
turn: boolean
Board()
InizializzaCampo():void
getBoard():Piece[][]
check(Point,Point):boolean
checkMate():boolean
validMove(Point,Point):boolean
movePiece(Point,Point):void
getPieceByIndex(Point):Piece
ControlloCollisioni(List<Point>):boolean
getValidMoves(Point):List<Point>
registerMove(Point,Point):void
findKing(PieceColor):Point
castle(Point,Point):boolean
isSquareAttacked(PieceColor,Point):boolean
enPassant(Point,Point):boolean
pieceIndex(Piece):Point
promotePawn(Point):void
-board
-enPassantPawn
0..*
0..1
<<Java Class>>
Piece
pieces
imgWidth: int
immagine: JLabel
mosso: boolean
Piece()
getColore():PieceColor
GetImage():JLabel
LoadImage(String):void
MovePezzo(int,int):void
MovePezzo(Point):void
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>
SetMosso():void
moved():boolean
#Colore
0..1
<<Java Enumeration>>
PieceColor
pieces
WHITE: PieceColor
BLACK: PieceColor
PieceColor()
<<Java Class>>
Bishop
pieces
Bishop(PieceColor)
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>
<<Java Class>>
Horse
pieces
Horse(PieceColor)
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>
<<Java Class>>
King
pieces
King(PieceColor)
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>
<<Java Class>>
Pawn
pieces
Pawn(PieceColor)
ValidaMossa(Point,Point):List<Point>
ValidaPresa(Point,Point):boolean
GetAllPossibleMoves(Point):List<Point>
<<Java Class>>
Rook
pieces
Rook(PieceColor)
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>
<<Java Class>>
Queen
pieces
Queen(PieceColor)
ValidaMossa(Point,Point):List<Point>
GetAllPossibleMoves(Point):List<Point>

As we see, every specific piece class incapuslates the rules that go for this particular piece only, while the board implements general rules of the game. What we have achieved with this architecture is a very best of object-oriented-model-like chess implementation i could think of, altough maybe it's not most straghtforward one. Infact we obtain the pieces implementation is highly independent from the chessboard, altough they are not that reusable, infact the board and the pieces are tightly linked in logical way.

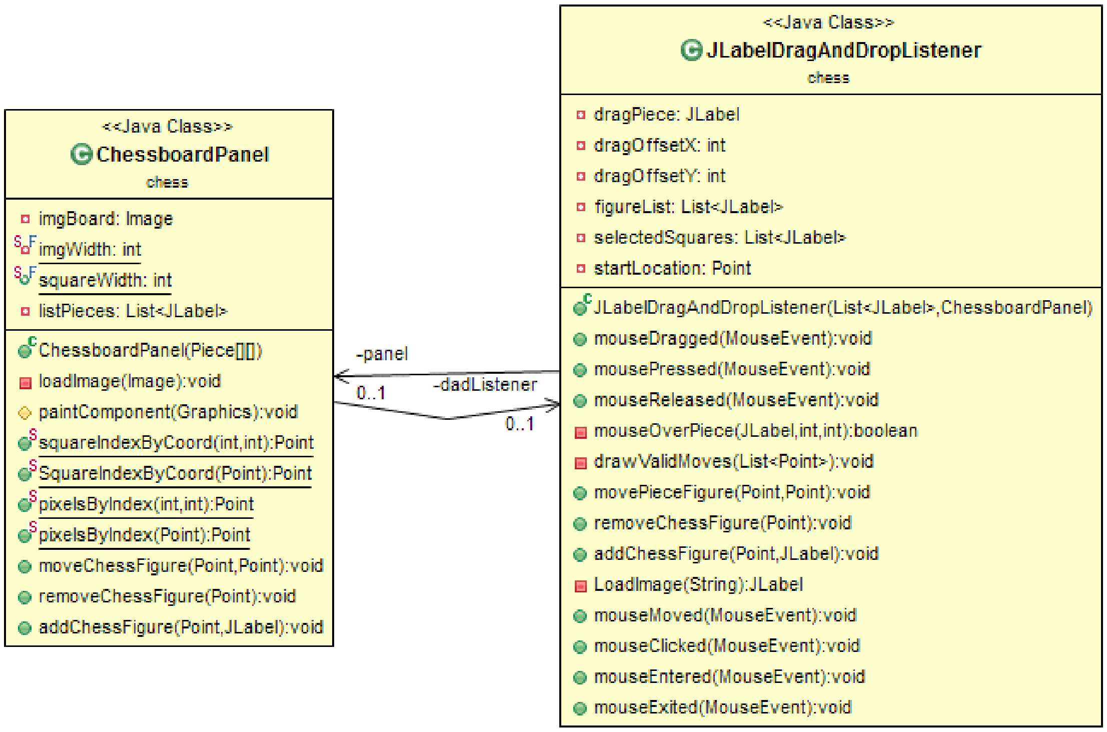
ChessboardPanel

Let's move to the ChessboardPanel class. This one inheritates from JPanel and is actually used in the MainWIndow class which on it's own inheritates from JFrame, but all the view logic is implemented in ChessboardPanel class. Let's begin by saying that it has some static general use method like pixelsByIndex and squareIndexByCoord that could be implemented in every other class, but i just chooesd this one. The constructor is not that good, it takes the 8x8 piece matrix, scrolls it over and for every not null cell, creates a new JLabel which stores the figure and paints it on the corresponding coordinates. And it also stores the JLbaels in a list, to keep track of them, and eventually remove them in the future. Definetly could have done better, quite sure the panel it self gives me the access to all the JLabels i've added to it, but it just gives me ALL of them, and i only need the figures, not the board JLabel for examle. Probably could have usen a tag property or something anyway, but just had no time for thinking it over so i kept this lame implementation. A very important member is JLabelDragAndDropListener dadListener. It handles all the interaction with the user, i'll look at it's implementation next, but in the ChessboardPanel constructor I just instatiate it (the listener i mean), and add it to the panel. Methods paintComponent and loadImage are the ones i found on internet just by googling for "showing an image in jlabel" or something like that.

JLabelDragAndDropListener

Next, the JLabelDragAndDropListener. It extends MouseListener and MouseMotionListener. It makes all the dirty job when it comes to the interaction with the users mouse. It also call's the Controller methods,that get redericted to the Board. It's constructor necessitates of the list of all JLabels that are added by the ChessboardPanel and the pointer to the parent panel itself. Now that's ugly, for the first parameter, a better solution would be to make the listeners constructor to add all the figures, not the ChessboardPanel. Then the ChessboardPanel wouldn't need the list of JLbaels at all. Second, almost certainly the listener by default has a pointer to it's parent, and i don't need to pass it at all. The drag 'n' drop logic itself is something i've looked for on the internet and i've made some (little) copy paste just to save time. When the user presses the mouse, we controll if the square he pressed on contains a piece, if it is the case, we store the piece in dragPiece, we store it's location in the startLocation and we ask the Controller to give us the coordinates of all the possible moves for the piece, and we just highilght the squares that are returned by the Controller. As the user moves the mouse, while it's still pressed, we move the dragPiece. When user releases the mouse, we call the Controller to check if the move user has just performed is legal. If it is, we leave the piece at the current position, if it is not, we return it to the startLocation. We don't need all of the methods required by the MouseListener and MouseMotionListener, so some of them are left blank.

Thats the view scheme



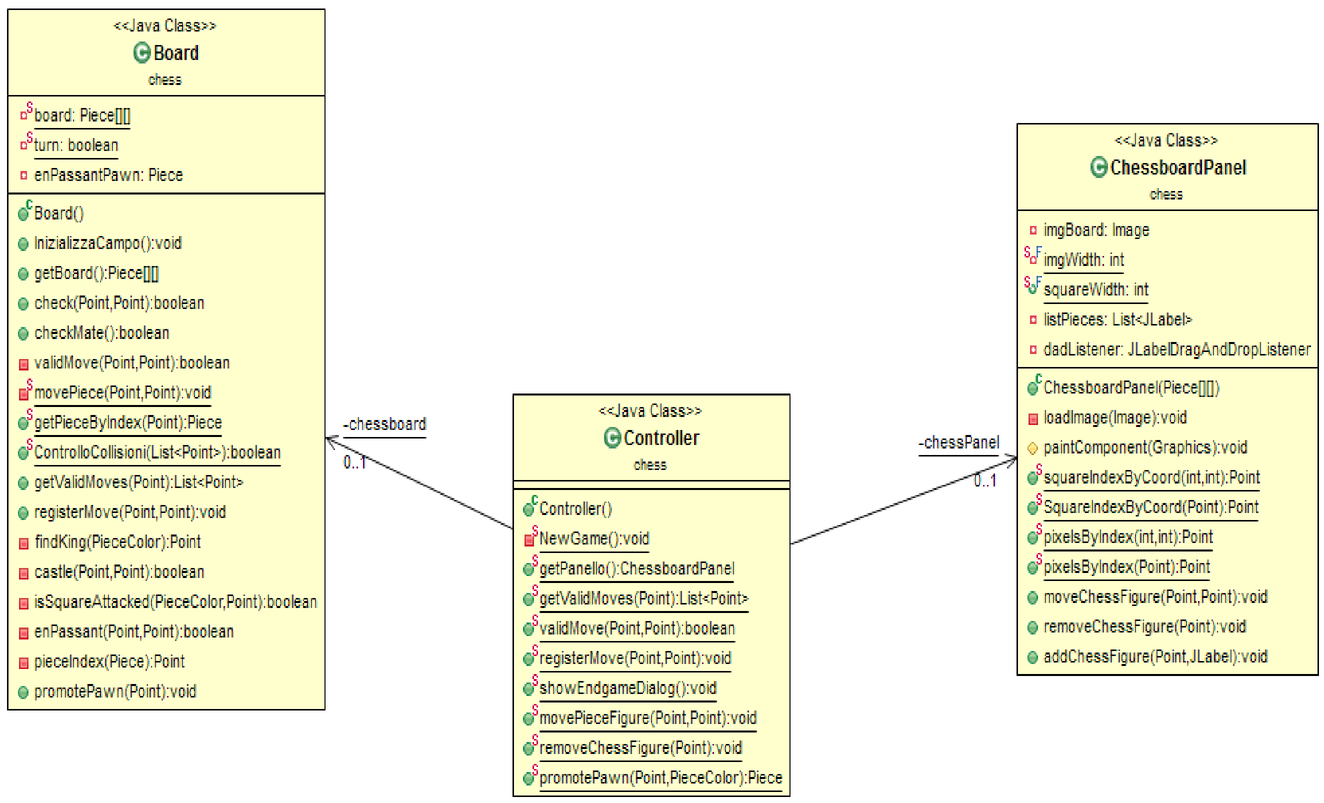
<<Java Class>>
ChessboardPanel
chess
imgBoard: Image
imgWidth: int
squareWidth: int
listPieces: List<JLabel>
ChessboardPanel(Piece[][])
loadImage(Image):void
paintComponent(Graphics):void
squareIndexByCoord(int,int):Point
SquareIndexByCoord(Point):Point
pixelsByIndex(int,int):Point
pixelsByIndex(Point):Point
moveChessFigure(Point,Point):void
removeChessFigure(Point):void
addChessFigure(Point,JLabel):void
-panel
0..1
-dadListener
0..1
<<Java Class>>
JLabelDragAndDropListener
chess
dragPiece: JLabel
dragOffsetX: int
dragOffsetY: int
figureList: List<JLabel>
selectedSquares: List<JLabel>
startLocation: Point
JLabelDragAndDropListener(List<JLabel>,ChessboardPanel)
mouseDragged(MouseEvent):void
mousePressed(MouseEvent):void
mouseReleased(MouseEvent):void
mouseOverPiece(JLabel,int,int):boolean
drawValidMoves(List<Point>):void
movePieceFigure(Point,Point):void
removeChessFigure(Point):void
addChessFigure(Point,JLabel):void
LoadImage(String):JLabel
mouseMoved(MouseEvent):void
mouseClicked(MouseEvent):void
mouseEntered(MouseEvent):void
mouseExited(MouseEvent):void

As we can see, ChessboardPanel owns an instance of JLabelDragAndDropListener and it exposes some public methods to allow the controller to communicate with the listener by calling the methods of ChessboardPanel. Also, the JLabelDragAndDropListener needs to keep reference to his parent ChessboardPanel, but that really isn't that neccessary, it's just a bad implementation of mine.

Controller

The Controller class is a static class that is meant to link the view and the model. It has a static instance of ChessboardPanel and of Board so it can freely rederict calls from the other classes, to the public methods of those two instances.

Thats the controller scheme


<<Java Class>>
Board
chess
board: Piece[][]
turn: boolean
enPassantPawn: Piece
Board()
InizializzaCampo():void
getBoard():Piece[][]
check(Point,Point):boolean
checkMate():boolean
validMove(Point,Point):boolean
movePiece(Point,Point):void
getPieceByIndex(Point):Piece
ControlloCollisioni(List<Point>):boolean
getValidMoves(Point):List<Point>
registerMove(Point,Point):void
findKing(PieceColor):Point
castle(Point,Point):boolean
isSquareAttacked(PieceColor,Point):boolean
enPassant(Point,Point):boolean
pieceIndex(Piece):Point
promotePawn(Point):void
-chessboard
0..1
<<Java Class>>
Controller
chess
Controller()
NewGame():void
getPanello():ChessboardPanel
getValidMoves(Point):List<Point>
validMove(Point,Point):boolean
registerMove(Point,Point):void
showEndgameDialog():void
movePieceFigure(Point,Point):void
removeChessFigure(Point):void
promotePawn(Point,PieceColor):Piece
-chessPanel
0..1
<<Java Class>>
ChessboardPanel
chess
imgBoard: Image
imgWidth: int
squareWidth: int
listPieces: List<JLabel>
dadListener: JLabelDragAndDropListener
ChessboardPanel(Piece[][])
loadImage(Image):void
paintComponent(Graphics):void
squareIndexByCoord(int,int):Point
SquareIndexByCoord(Point):Point
pixelsByIndex(int,int):Point
pixelsByIndex(Point):Point
moveChessFigure(Point,Point):void
removeChessFigure(Point):void
addChessFigure(Point,JLabel):void

From this UML it seems clear that the model and the view are fully separated entities that communicate with each other by the means of the static Controller class. Altough it's the Controller class to own the instances of Board and ChessboardPanel, it does a very little job, which consist mostly in providing public methods to interface the calls to the instances of Board and ChessboardPanel.

3. Details

The chess rules are implemented in this way : in general, when the user choeses a piece to move, JlabelDragAndDropListener asks for legal moves. The controller redericts the call to the Boards getLegalMoves method. This method works this way :

- Calls pieces getAllPossibleMoves which returns of all possible moves the piece could hypothetically make from current position
- Iterates the list, and for each move calls the Boards validMove method. If the move isn’t valid, then it eliminates it from the list
- Because validMove doesn’t perform the check condition check, it iterates another time to remove all the possible moves that would expose the king under attack, as those are illegal.
- It returns the list of the moves that are legal.

When the boards validMove is called, it performs this actions :

- checks if pieces color is coherent with the turn that it keeps track of
- checks if the destination square doesn't contatin a piece of same color
- asks the particular piece if the move the user want to perform is ok for the concrete piece
- if it is, the Board also gets from the piece the list of all the squares it needs to cover to get to the destination square, except the start square and the destination squares themselves, so the board checks if there are collisions on the way of the piece. The horse piece just returns an empty list because it doesn't care if there are collisions.
- then it checks for some particulare conditions as castling and en passant which i will explain further
- it then checks if the move is valid by calling **pieces** validMove that just tells if the move is legal for this **type** of piece
- if we're still okay, we perform the collision checking which is very straightforward. We do the collision checking and the move validation at the end of the method just because we wan't to check we're in presence of some special condition like castling that require us to know some board information also, and the pieces validMove method just doesn't consider the board at all, so we would get some buggy behaviour.

The en passant mechanism works as following :

- Each time we move the pawn by two squares, we store it in the enPassantPawn variable and keep it for just one turn
- The enPassant(Point source, Point destination) method checks if enPassantPawn != null, and if it is, it checks if the pawn located at the source position has on it's righ or on it's left an enemy pawn that has just made a double move
- When we register a move, we check if the en passant has just occured, because we need to inform the controller to remove the eaten pawn figure as this is a particular case, and we also remove the eaten pawn manually from the logical board. Then we put null in the enPassantPawn variable.

The castling works as following :

- Boards validMove method calls the castle method.
- The castle method checks if the piece is a king and if it never moved before
- Then we check if the rook we're interested in never moved before, and if it can move to the position just near the king, this way we also perform the control that the squares it needs to cover, perform no collisions.
- Then we also need to check that the squares that the King needs to cover when performing the castling, are not attacked by any enemy piece. So we call the isSquareAttacked method by passing it some hardcoded parameters.
- If all this conditions hold true, we can perform castling. As with en passant, in the registerMove method we need to recognize that the castling has occured and do some manual job as moving the rook figure ecc.

Check. The check method is pretty straightforward. But it's use is not, see the validMove method explanation for the use of Check. What we want from this method is to give it 2 parameters, source and destination, and get a boolean result which should indicate if the check **would** occur if we would move the piece from source to destination. So what we do is :

- We backup some information as the turn, because i need to change it (i do this also in isSquareAttacked method, because we perfromf calls to validMove method in theese methods, and the validMove performs some turn controlls that are not good for hypothesising) and the piece that is eventually stored in the destination square (doing this because i'm going to move the piece from source to the destination, and by doing so i will erase everything was stored in the destination square before, and that's also not good for hypothesising)
- We store the kings location as we'll need it
- Then we just iterate through all over the board to check if there is some piece that could attack the destination square. We do so by calling the validMove method with iteration indexes and king location as parameters.
- Then we just perform the roll back in any case

Checkmate. The checkmate is very straightforward. Basicly we just :

- Iterate through all over the board, and for each piece of the color of interest we call the getValidMoves. If for no piece this method returns a list that is not null, means there are no legal moves at all, so the checkmate has occured.

As far as i know, and i play chess pretty often, i've implemented all the standard chess rules, except the one that gives player the chance to choose which piece transform the promoted pawn into, i just promote it to the queen by default as it is the defaul choice almost in every case.

I don't think i've implemented any partycular OOP pattern, at least not consciously, altough i've planned to implement the skin mechanism by the use of factory pattern.

4. Package organization

JavaChess.src.chess - contains all the code that has to do with the chessboard

JavaChess.src.pieces - contains the abstract Piece class, and all the specific piece Classes and also PieceColor enumerator.

JavaChess.src.images - contains all the images used by the application

5. Work subdivision

The project was entirely developed by me, Jevgenij Semionov

6. Testing

Applied no unit testing, just beta testing which showed up some bugs that i've elminated, altough not 100% sure that there are no more.

7.Conclusions

The developement of the application was mainly linear as i've ended up with the architecture i had in my mind all along. Altough the game is fully playable, i'm not happy with some decisions i've implemented, mainly with some redundant code, magic numbers and some parts of the code that are clearly inefficient, and i just can't consider it fully finished. Also because i didn't implemented the possibility for the user to choose which piece promote the pawn into, and i just promote it to the queen by default, as do all the players in the world, but still that's one rule i didn't implement because of lack of time, altough i've started to implement the PawnPromotionDialog. I'd also like to give user a chance to choose between the horizontal and vertical perspective. The program also remains in memory after closing it, maybe need to call a dispose method or something. In future i consider to sometimes work on this project and use it as my default programm for playing chess with my friends.