**Design Document**

***The Grandmaster Pawns***

Advait Maybhate, Ben Brunt, Isabel Shu and Lucy Liu

****

**Requirements:**

Our Chess project allows two users to play a game of chess with a graphical interface. Our game will include the basic rules of chess such as the grid locations where the pieces are allowed to move to, forcing the players to take alternating turns, check and checkmate, as well as pawn promotion. In addition, Kings will not be able to put themselves into danger (check). By selecting a piece on their turn, users will be able to identify where their piece can move to based on a red and green colour scheme (red locations highlight spots where the selected piece can attack an opponent’s piece and green locations highlight spots to which the selected piece can move). Each player will take turns to choose their desired move. Rules of chess that will not be implemented in our program include stalemate, en passant, and castling (see “Future Features” section for more details).

**Design:**

As you start the program, a JFrame will pop up, filled with a 2-D array of JButtons on a 8 x 8 grid. These JButtons will have the appropriate images of the chess pieces’ starting positions displayed on the checkered grid. As mentioned, the grids will change accordingly based on movable and threatening locations. All these graphics are to be completed in the “ChessBoard” class.

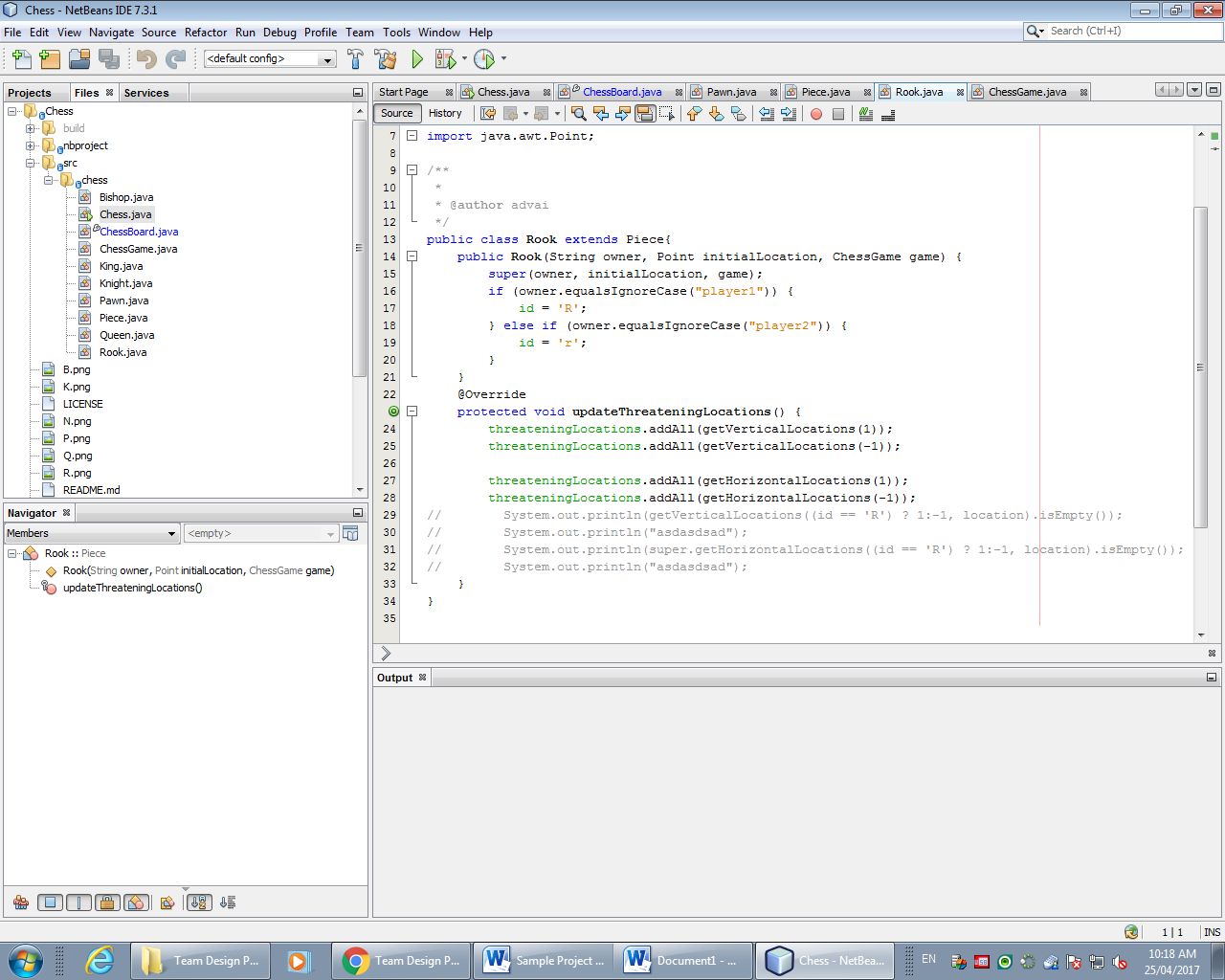
In the program, we created the superclass “Piece” containing all the attributes that all chess pieces have in common such as movable locations, threatening locations, the current location of the piece, the type of piece, as well as the player. In here, various methods are created to update the movable/threatening locations. According to the chess piece, appropriate methods will be called (these methods are overridden in the individual classes).

Additionally, all of the individual chess pieces have their own class, extending “Piece”. All classes define the ID of the piece and update the threatening locations by adding possible locations to an ArrayList, namely “threateningLocations”. This is done via calling appropriate methods from the “Piece” class according the legal moves that the piece is allowed to move to. For example, the bishop class would only call “getDiagonalLocations”.

However, while this works for most pieces, the Pawn class and the King class are exceptions. The pawn has a special first move attribute as it can move two spaces. Unlike any other piece, it also has a different set of points for threatening and movable locations, as it moves vertically but attacks diagonally. For the King, it is crucial that it does not move itself into a location where it is put into danger (check). Additionally, the two Kings cannot threaten each other, or else the game would not be resolved.

**Implementation:**

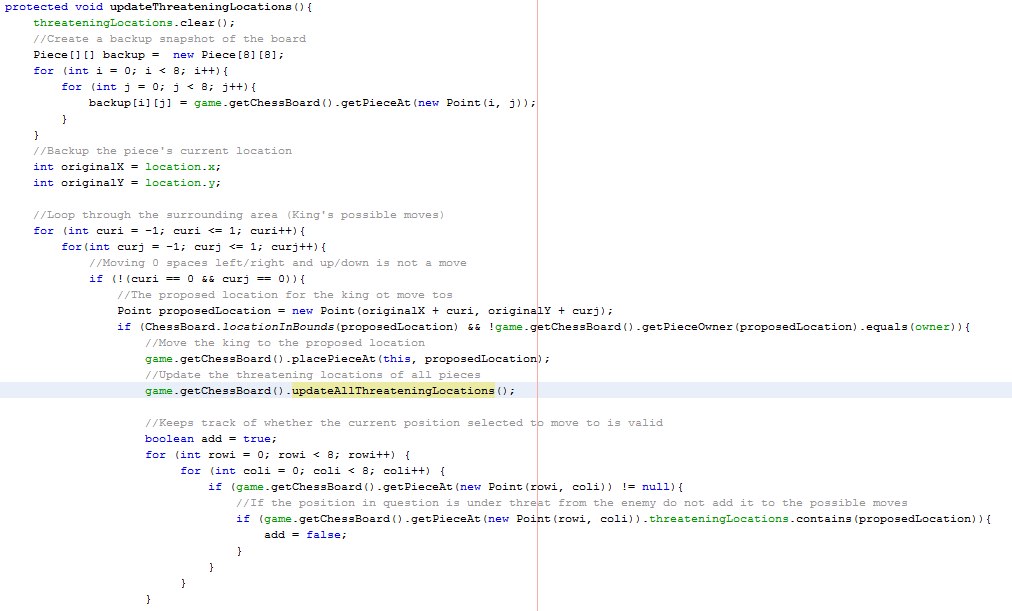
To create the graphics for our chess board, we created a JFrame, containing a JPanel for the board itself. This board consists of a 2D array of JButtons. We then loaded the appropriate images onto the buttons by running through the array. By making the grid of buttons, it is easier for us to gather the location of the piece using an ActionListener compared to gathering the x and y coordinates of the mouse location. The ActionListener allows us to detect when the user clicks on a JButton, and where the JButton is located on the grid.

To implement our program, we used our superclass “Piece” to declare attributes that all chess pieces have in common mentioned above. In here, we have created methods consisting of updating and showing the movable/threatening locations through “getVerticalLocations”, “getHorizontalLocations”, and “getDiagonalLocations”. Each individual piece then has its own class, extending “Piece”. In each class, the piece ID is established as a upper or lower case letter representing both the piece and player. The following is an example of establishing the ID in the Rook class: 

All classes also contain an “updateThreateningLocations” method. This is where the the methods aforementioned in “Piece” will be called to add to the arraylist possible locations the piece can move to.

To have a different set of points available for the pawn -- differentiating between movable and threatening locations -- two methods are used in a similar manner. In the movable locations method, the points added to the arraylist based on the current location of the pawn piece can only be one step forward, unless there is already a piece in front of it. If the piece however is established as a first move, also unique to the pawn piece, then it is allowed to move up to two steps forwards. In the threatening locations method, it checks for the locations one step diagonal to the current location to see if there are any pieces from the opponent that it could attack.

In the King class, there are multiple exceptions that the program must overcome. Specifically, the King cannot move anywhere where it will be threatened (cannot move into check). In its “updateThreateningLocations” method, it creates a “backup” 2D array that essentially back ups the entire state of the current chess board. Then, imagining the current location of the King to be (0,0), the method will check all its surrounding locations, running through a double for loop that runs from -1 to 1. Using nested if statements, the King piece is moved into all its possible locations. This is done without updating graphics, so it is not visually seen. In each of its locations, it first sees if the own player’s piece is there. It also checks to see if moving to that location will place the King under any sort of threat. For example, if moving the King allows it to be attacked in the opponent’s next move, then moving there is not an option for the King (see code below for a more detailed explanation of this concept).



Additionally, one must check if the location proposed is closer than two spaces from the other King since the two Kings cannot be beside each other (this will cause a conflict and the game cannot be resolved). After checking these conditions, all viable locations will be added to the “threateningLocations” ArrayList, allowing the graphics to be updated accordingly.

**Testing:**

In the early stages of programming, we use a text-based version of Chess to ensure that our code does what it is supposed to. During if statements in our code, unique print statements are used to ensure certain scenarios are reached and used. Following the completion of our program, testing consists of playing our game of Chess when we have the graphics and classes working. Through this type of testing we will try to catch any holes in the code that may exist by attempting to move pieces in positions where they should not be allowed. We will also look at the threatening locations displayed to see if they are accurate based on the situation.

**Installation:**

We understand that many users may not have JDK 7 (Java Standard Edition Development Kit 7) which is why we have created an executable jar file that is easy to run. The user will need to download this jar file (ideally from GitHub) and will need JRE 7 (Java Standard Edition Runtime Environment 7). Next, they can simply double click on the jar file in order to execute the program (advanced users can run this program via command prompt with the “java -jar” command).



Running the game via command prompt (using the jar file)

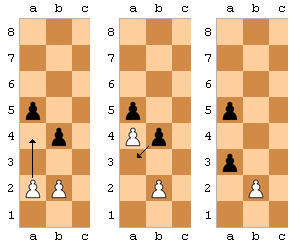
**Maintenance:**

As we go through our testing phase, we will make notes about areas of the game that should be improved allowing us to know which areas of our code that should be improved. This will enable us to continually update and improve our program to work as well as possible.

**Future Features:**

The following list encompasses the features of chess that we would like to implement in the future:

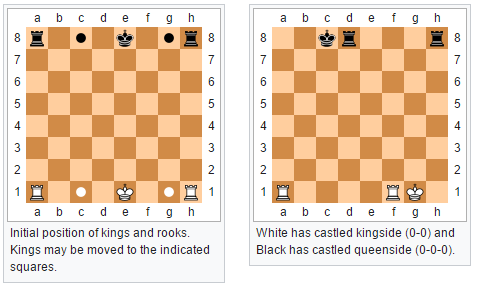
* En Passant
  + A special move for pawn capture which involves pawns moving 2 rows forward if the enemy pawn could have captured the pawn had it only moved 1 row forward. In this scenario the enemy would still be able to capture this pawn, according to the rules of chess. However, this rule has not been implemented in our game.



A diagram demonstrating en passant

*Image Source:* [*https://en.wikipedia.org/wiki/En\_passant*](https://en.wikipedia.org/wiki/En_passant)

* Castling
  + Castling is not currently implemented in the game. Castling involves the “switching” of the rook and the king to allow for the king to be better protected, without wasting many moves.



Demonstration of castling

*Image Source:* [*https://en.wikipedia.org/wiki/Castling*](https://en.wikipedia.org/wiki/Castling)

* Preventing illegal moves
  + Currently, players are allowed to move their pieces in such a manner that they can put their own king into check, allowing their opponent to take their king on the next move. Currently we have added a dialog that informs the player that their king has been taken and their opponent has won. Ideally, we would prevent the piece in the first place, preventing an illegal move (putting one’s own king into check).
* Stalemate
  + A stalemate is caused when a player cannot make a legal move however he/she is not in check. This results in a draw between the players. However, this is not currently implemented in the game.

**Completion Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Team member(s) responsible** | **Target completion date** | **Status** |
| **Plan the overall design** | **All** | 4/18 | done |
| **Write this project plan** | All (See below) | 4/25 | done |
| * Requirements, Design, Implementation, Testing, Installation, Maintenance | Ben, Lucy, Advait | 4/27 | done |
| * Completion Plan | All | 4/25 | done |
| **Code the Chess app** | **All** (see below) | 4/25 | done |
| * Code the Chess class | Advait | 4/21 | done |
| * Code the Chess Game class | Advait | 4/21 | done |
| * Code Bishop Class | Lucy, Isabel | 4/22 | done |
| * Code King Class | All | 4/23 | done |
| * Code Knight Class | Ben, Advait | 4/21 | done |
| * Code Pawn Class | Advait | 4/20 | done |
| * Code Queen Class | Lucy, Isabel, Ben | 4/21 | done |
| * Code Rook Class | Advait | 4/21 | done |
| * Code Piece Class | Advait, Ben | 4/20 | done |
| * Chess Board class (includes graphics) | Lucy, Isabel, Advait | 4/26 | done |
| Testing and debugging | All | 4/27 | done |
| Write internal documentation | All | 4/25 | done |
| Write external documentation | Lucy | 4/27 | done |
| **Submit to Mr.Wilhelm** | Advait - Documents and code  All - Peer evaluation forms | 4/30 | done |