

Mathew George's 2008 Chess Simulator Dossier

Table of Contents

A1: Analyzing the Problem	Page 2
A2: Criteria for success	Page 26
A3: Prototype Solution	Page 28
B1: Data Structures	Page 42
B2: Algorithms	Page 46
B3: Modular Organization	Page 59
C1: Using good Programming Style	Page 62
C2: Usability	Page 103
C3: Error Handling	Page 106
C4: Success of the Program	Page 107
D1: Including an Annotated Hard Copy of the Test Output	Page 110
D2: Evaluating Solutions	Page 117
D3: Including User Documentation	Page 120
E: Holistic Approach to the Dossier	Page 125

A1: Analyzing the Problem

Introduction

I have always wanted to build games for computers but never had a chance either from lack of available time or, until recently, lack of skill at programming. Beyond that, chess has always been one of my favorite games because it requires the player to think well in advance at what they are going to do while still allowing people who are impulsive the chance to play, at the very least, with some success. By making a chess simulator I had hoped to improve my overall understanding of the game and, as a result, improve my personal skill at the game.

Current Systems

I have checked 3 different current systems for playing chess. The first is the website <http://www.chess.com> one of the largest online chess communities providing numerous services at no cost to users. The second is a stand alone chess simulator that I found on the previous site posted by one of the users by the name of SonofPearl and I am assuming is also the creator of the game. The third is a basic over look of the rules of chess and a bit on the history of chess history and rules were found on chess.com.

www.chess.com

This site is one of the largest chess sites around providing both computer and human opponents from its large player base. Also, the site has sections dedicated to enabling players to improve with tutorials and training sections provided.

Interface and Functions

1) Main Menu going from left to right, option allows players to start matches (Play), introduces new and experienced players to resources to improve their skill in the game (Learn), provides information on books and shops near the player for more information on chess (Shop), talk with other people on the board on a variety of subjects (Forums), new articles on chess (News), upcoming events such as tournaments (Events), information on other members of the site (Members), chess clubs across the world and online (Clubs), information on a range of subjects within the chess world (Directories), free chess games available for download (Downloads) and access to fun things related to chess (Fun).



Figure 1: Main menu

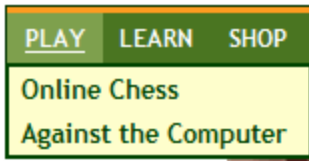


Figure 2: Play Menu

2) **Play** Only two options provided online chess which allows the user to find an opponent of similar skill level to them self and challenge them to a game. The other option, against the computer, allows the player to play games against a computerized opponent

3) **Learn** The chess mentor allows users to use an application that checks the user's moves against a database of possible moves and gives feedback on the player's choice allowing them to improve over time. The Chess articles is a database of recent articles on chess related topics and the effects chess have



on people. Puzzle of the day is a daily challenge that gets the player to decide on the next few moves the player should make to win the match. Chess rules and basics provide information on the general game of chess that all players should know before playing a match. Next, the Chessopedia has a list of all the common strategies players employ. The game analysis page allows players to ask other players where they went wrong to make better choices in future games. Finally, find a coach section is there to help you find people that may be able to assist in improving your game.

4) **Shop** The shop menu in general is to assist in finding

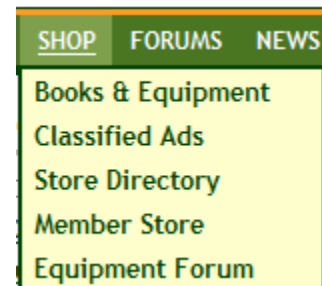


Figure 4: Shop Menu

information on chess i.e. strategy books or equipment to play the game ranging from new to second hand.

5) Forums Access to the posts you have flagged ranging from most hot topic of the day to topics the user has been involved in.



Figure 5: Forums Menu

6) Members Links to locations where users can find information on specific players or popular user blogs/photos last of which is a list of where the majority of players are from.

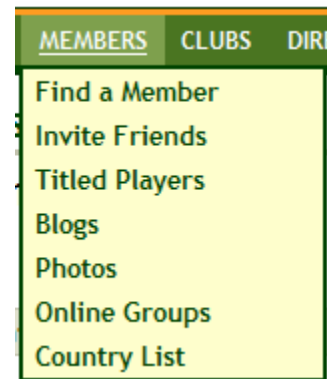



Figure 6: Members Menu

7) Clubs Find information on chess clubs near you or online clubs giving the clubs name, logo, small description of the club, the number of members and date formed.



[Chess.com Fans](#)

Love the site? Join this group for insider news, forums, and more! [Read more »](#)

International 

Members:

169

Formed:

Sep 03 2007

Figure 7: Online Club info example

8) Directories A range of information from places to shop including detailed information where coaches can be found or produces can be purchased to videos of recent matches.

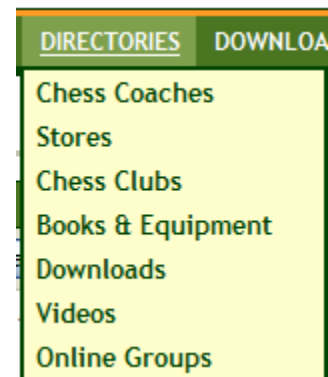


Figure 8: Directories Menu

9) Downloads Provides downloads of chess freeware that can be used to improve a players game includes the title of the item, a short description, popularity, rating and more info button which leads to a more comprehensive page on the item in question.

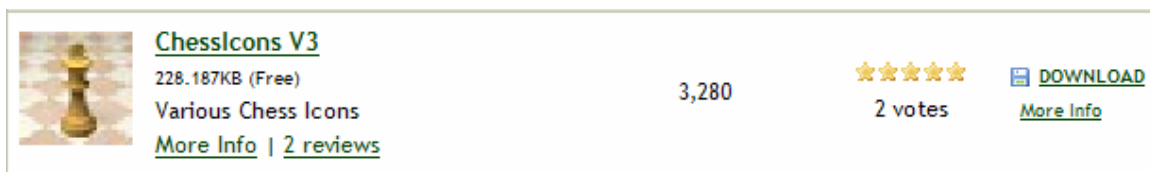


Figure 9: Example of information on downloads

10) Fun Links to various videos and other almost trivia information related to chess.

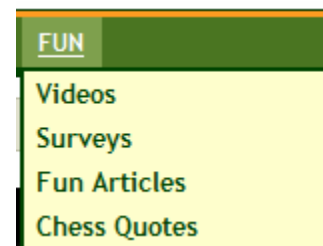
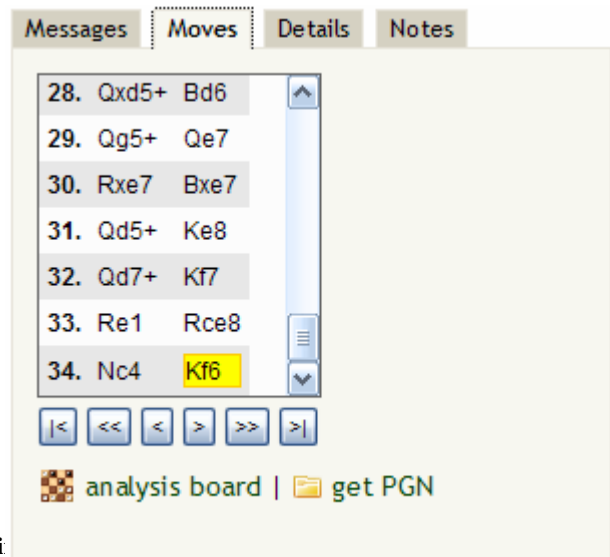
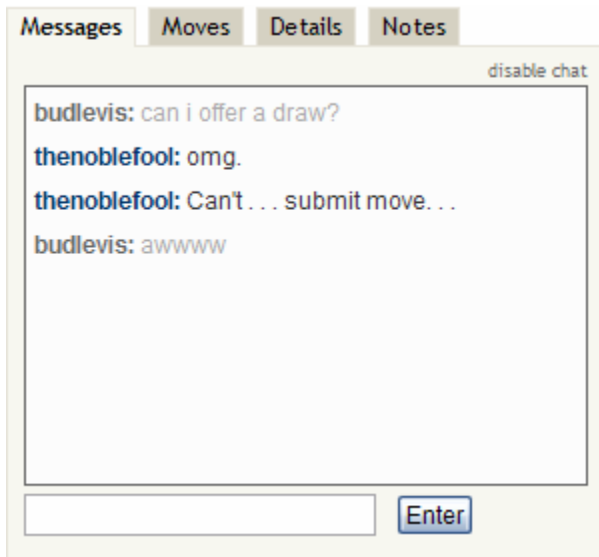


Figure 10: Fun Menu

11) Interface the actual interface of the game is simply the board and graphical representations of the pieces as well as a chat window on the right to allow the players to talk about the game there playing.



Figure 11: Game in Progress



Messages Moves **Details** Notes

White: [thenoblefool](#) - 2 days 22 hours left
 Black: [budlevis](#) - 3 days left
 Unrated

Flip board

Captured Pieces:

14	
23	

Let's Play!

Start: 19 Dec 2007 | Time: 1 move per 3 days

Messages Moves Details **Notes**

PRIVATE NOTES! Nobody else can see these notes - they are for your game ideas.

OFFER DRAW | RESIGN

DreamChess

This program is a fan made chess simulator with a computer opponent ranging from difficult 1-8 and the option to have a player vs. player game. It has been downloaded over 9 thousand times as of this review.

Interface and Functions: Custom menus and dialogue boxes made by the original maker

1) **Main Menu** Presents the user with the option to start a new game (New Game..), load a previously saved game (Load Game..), to change the appearance of the pieces, board and background during a match (Select Theme), or the quit the game (Quit),

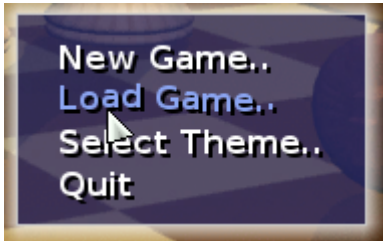


Figure 1 Main Menu

2) **New Game** Prompts the user to start the game after first selecting weather they want to play against the computer or a player (and if computer weather they want to go first or second) below that is the difficult range of the computer controlled player with the player choosing the difficulty between 1 and 8 (1 being easiest, 8 being hardest). Finally, the game offers the player if they want to go back to the main menu.

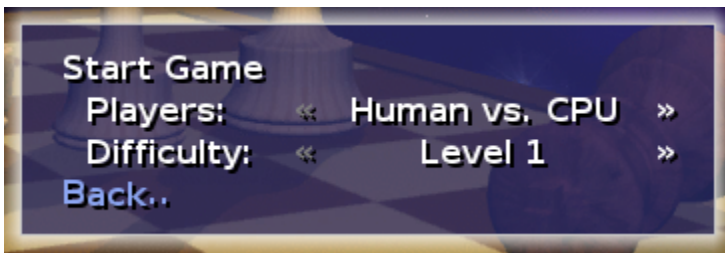


Figure 2 New Game Menu

3) Load Game The load game menu allows the player to call games that were saved during play. Up to 15 games can be saved at a time showing the position of the pieces, the difficulty, who is controlling which side and the time it was saved at.



Figure 3 Load Saved Menu

4) Select Theme allows the user to change the background and pieces to match a certain look even allowing the user to mix and match different pieces with different backgrounds and general set up of the theme.



Figure 4 Theme window



Figure 5 Sketch board theme

5) In Game Menu By right-clicking the mouse or hitting the escape key, the user can pull up the in game menu which provides the user with the options to first return to the game (self explanatory). The save game option which is nearly identical to the load menu

except the user saves the games rather than loading them. Finally is the more options choice which leads to another menu.

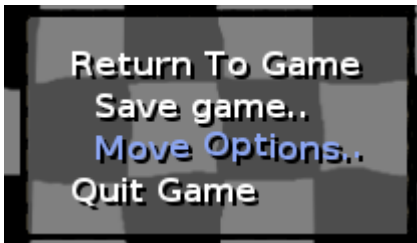


Figure 6 In-game menu

6) More Options This sub-menu of the in game menu allows the user to take back all the moves he has made so far in the game (the computer automatically takes back an equal number of moves as well). Force the computer to makes its move which is only useful on the higher difficulties when the computer tends to take its time making a move. The last two options are the view previous and view next moves which allows the user to go through each move that BOTH sides have made one at a time i.e. if it's the second move it will show where the second players piece was last turn then yours.

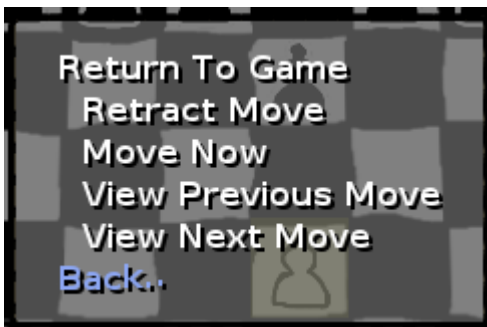


Figure 7 Move Options menu

Chess

The board game chess is one of the most widely played games around in the world currently. Modern chess that is played by most people originated from Southern Europe in the 15th century based off similar games played in the middle-east.

Rules Taken from <http://www.chess.com/article/view/chess-rules--basics>

Origins of Chess

The origins of chess are not exactly clear, though most believe it evolved from earlier chess-like games played in India almost two thousand years ago. The game of chess we know today has been around since the 15th century where it became popular in Europe.

The Goal

Chess is a game played between two opponents on opposite sides of a board containing 64 squares of alternating colors. Each player has 16 pieces: 1 king, 1 queen, 2 rooks, 2 bishops, 2 knights, and 8 pawns. The goal of the game is to checkmate the other king. Checkmate happens when the king is in a position to be captured (in check) and cannot escape from capture.

Starting a Game

At the beginning of the game the chessboard is laid out so that each player has the white (or light) color square in the bottom right-hand side. The chess pieces are then arranged the same way each time. The second row (or rank) is filled with pawns. The rooks go in

the corners, then the knights next to them, followed by the bishops, and finally the queen, who always goes on her own matching color (white queen on white, black queen on black), and the king on the remaining square.

The player with the white pieces always moves first. Therefore, players generally decide who will get to be white by chance or luck such as flipping a coin or having one player guess the color of the hidden pawn in the other player's hand. White then makes a move, followed by black, then white again, then black and so on until the end of the game.

Moving the Pieces

Each of the 6 different kinds of pieces moves differently. Pieces cannot move through other pieces (though the knight can jump over other pieces), and can never move onto a square with one of their own pieces. However, they can be moved to take the place of an opponent's piece which is then captured. Pieces are generally moved into positions where they can capture other pieces (by landing on their square and then replacing them), defend their own pieces in case of capture, or control important squares in the game.

The King

The king is the most important piece, but is one of the weakest. The king can only move one square in any direction - up, down, to the sides, and diagonally. Click on the ">" button in the diagram below to see how the king can move around the board. The king may never move himself into check (where he could be captured).

The Queen

The queen is the most powerful piece. If moved she can move in any one straight direction - forward, backward, sideways, or diagonally - as far as possible as long as she does not move through any or her own pieces. And, like with all pieces, if the queen captures an opponent's piece her move is over. Click through the diagram below to see how the queens move. Notice how the white queen captures the black queen and then the black king is forced to move.

The Rook

The rook may move as far as it wants, but only forward, backward, and to the sides. The rooks are particularly powerful pieces when they are protecting each other and working together!

The Bishop

The bishop may move as far as it wants, but only diagonally. Each bishop starts on one color (light or dark) and must always stay on that color. Bishops work well together because they cover up each other's weaknesses.

The Knight

Knights move in a very different way from the other pieces – going two squares in one direction, and then one more move at a 90 degree angle, just like the shape of an “L”.

Knights are also the only pieces that can move over other pieces.

The Pawn

Pawns are unusual because they move and capture in different ways: they move forward, but capture diagonally. Pawns can only move forward one square at a time, except for their very first move where they can move forward two squares. Pawns can only capture one square diagonally in front of them. They can never move or capture backwards. If there is another piece directly in front of a pawn he cannot move past or capture that piece.

Promotion

Pawns have another special ability and that is that if a pawn reaches the other side of the board it can become any other chess piece (called promotion). A pawn may be promoted to any piece. [NOTE: A common misconception is that pawns may only be exchanged for a piece that has been captured. That is NOT true.] A pawn is usually promoted to a queen. Only pawns may be promoted.

En Passant

The last rule about pawns is called “en passant,” which is French basically means “in passing”. If a pawn moves out two squares on its first move, and by doing so lands to the

side of an opponent's pawn (effectively jumping past the other pawn's ability to capture it), that other pawn has the option of capturing the first pawn as it passes by. This special move must be done immediately after the first pawn has moved past, otherwise the option to capture it is no longer available. Click through the example below to better understand this odd, but important rule.

Castling

One other special rule is called castling. This move allows you to do two important things all in one move: get your king to safety (hopefully), and get your rook out of the corner and into the game. On a player's turn he may move his king two squares over to one side and then move the rook from that side's corner to right next to the king on the opposite side. (See the example below.) In order to castle, however, it must meet the following conditions:

- it must be that king's very first move
- it must be that rook's very first move
- there cannot be any pieces between the king and rook to move
- the king may not be in check or pass through check

Notice that when you castle one direction the king is closer to the side of the board. That is called *kingside*. Castling to the other side, through where the queen sat, is called castling *queenside*. Regardless of which side, the king always moves only two squares when castling.

Check and Checkmate

As stated before, the purpose of the game is to checkmate the opponent's king. This happens when the king is put into check and cannot get out of check. There are only three ways a king can get out of check: move out of the way (though he cannot castle!), block the check with another piece, or capture the piece threatening the king. If a king cannot escape checkmate then the game is over. Customarily the king is not captured or removed from the board, the game is simply declared over.

Draws

Occasionally chess games do not end with a winner, but with a draw. There are 5 reasons why a chess game may end in a draw:

1. The position reaches a stalemate where it is one player's turn to move, but his king is NOT in check and yet he does not have another legal move
2. The players may simply agree to a draw and stop playing
3. There are not enough pieces on the board to force a checkmate (example: a king and a bishop vs. a king)
4. A player declares a draw if the same exact position is repeated three times (though not necessarily three times in a row)
5. Fifty consecutive moves have been played where neither player has moved a pawn or captured a piece.

Literature Cited

Chess.com Chess Rules & Basics Submitted by Chess.com Webmaster on Tue, 05/08/2007 at 3:44pm. Retrieved December 20, 2007, from the World Wide Web:

<http://www.chess.com/article/view/chess-rules--basics>

DreamChess V 0.1.0, A Playing Programs & Engines for chess by DreamChess Submitted by SonofPearl on Thu, 11/08/2007 at 2:19pm. Contact information could not be found. Retrieved December 20, 2007, from the World Wide Web:

<http://www.chess.com/download/view/dreamchess>

Chess.com Home site created by Chess.com Webmaster (actual name unknown) and created at an unknown date. Retrieved December 20, 2007, from the World Wide Web:

<http://www.chess.com>

Analysis

When I started on the problem, there were two possible routes I could take for output. I could either do a graphical user interface (GUI) or make it a command line interface. I quickly decided that GUI would be the better choice since games are supposed to be for the enjoyment of all and using a command line interface makes it easier for less computer savvy users to understand what they are supposed to do.

Additionally, I also thought about the esthetics of the two choices and considered that the GUI would be considerable more pleasing to the user's eye than the command line.

An example of an early design template for how I was planning to make the visuals of the board is below.

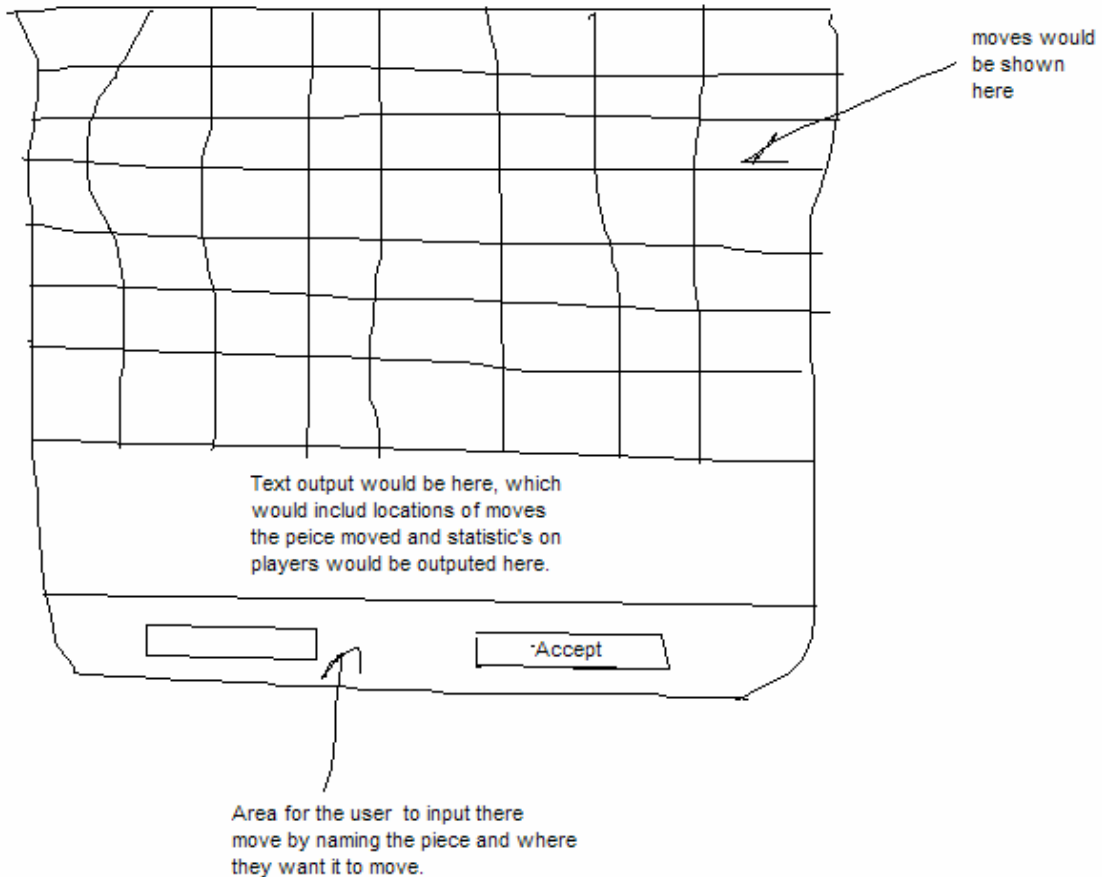


Figure 8: early design of interface

Alternatively, the grid at the top could be replaced in favor of a second text area outputting the whole board each time a move was made allowing players to see the progression of the game by scrolling back into previous moves or looking over the outputted moves allowing the players to have both a visual past and a coordinate moved versions of what happened.

After that I would have to decide if the game will require players to login in order to play a game. Requiring them to login will help keep track of statistics, but it will make

it more time consuming to play a game since two players will be required to login. I think it would be considerable easier on both me, as the programmer, and the end-user to set it up so that players are not required to login enabling a player to decide if they want their statistics to be updated by whether or not they have logged in.

Input and Output

The input could be done in two significantly different ways. Either have the player select the piece they want to move and where they want to move it using the mouse, which would only work with the graphical board not the text area, or have the player state with piece they want to move and where they would like to move it, would work with both text and visual board ways. For the text area input, it would likely be inputted by the user in a format similar to 'Piece', followed by a space, then the coordinates on the board where the player wants to move the piece. The overall position of all of the pieces will have to be kept in a two-dimensional array until they are accessed to be moved, shown visually or saved.

Beyond specific game input, there are also player profiles to load before games, if the players choose to, and profiles to save, not up to the player the profiles will auto-save when a game is saved or when a game ends. As for saving games, that should be do-able at anytime so long as the game is still going. For selecting the file to load it would be easiest to use a file chooser and combine that with random access file handling to allow only parts of the file to be used.

Systematic Analysis

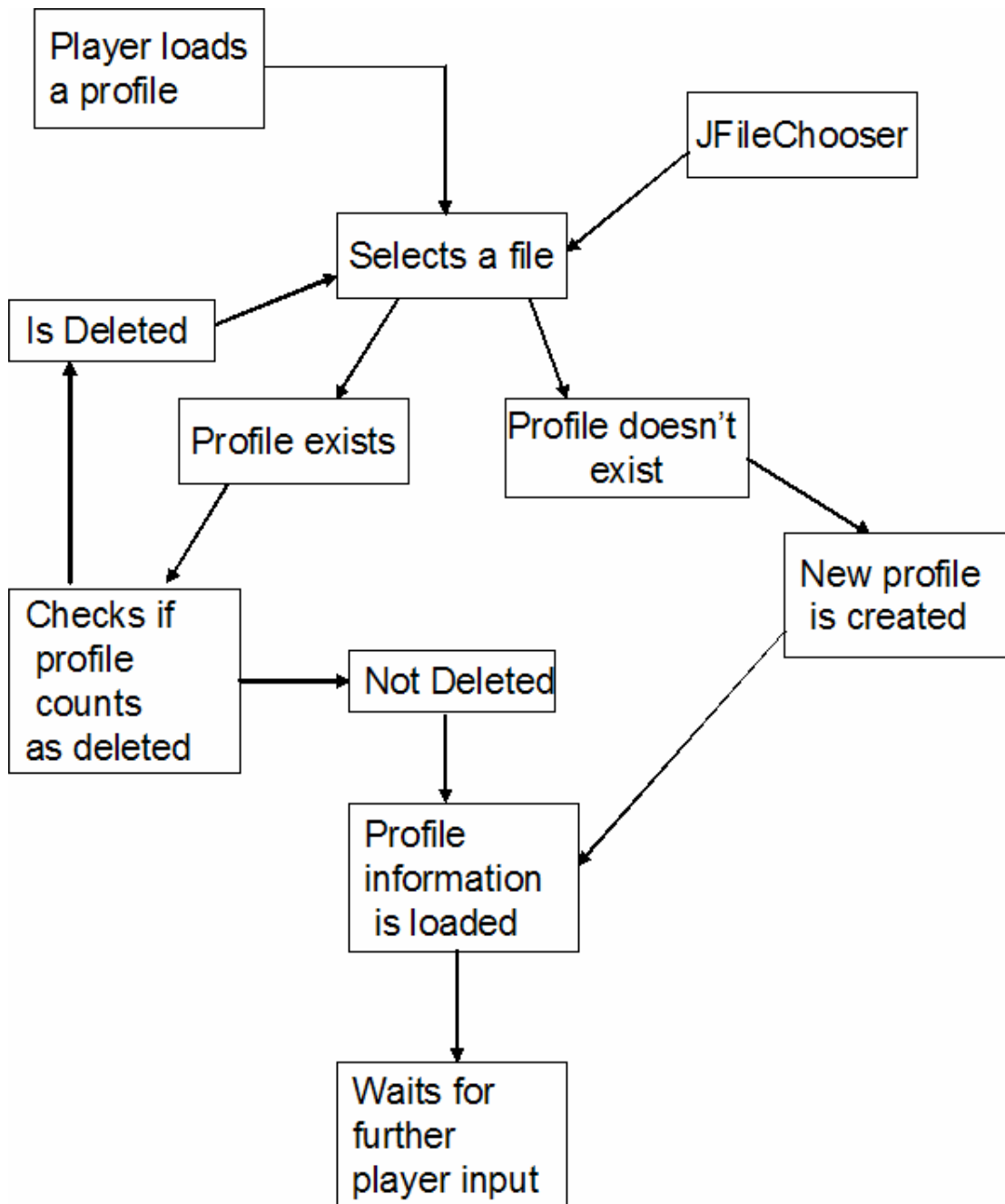


Figure 9: Loading profiles

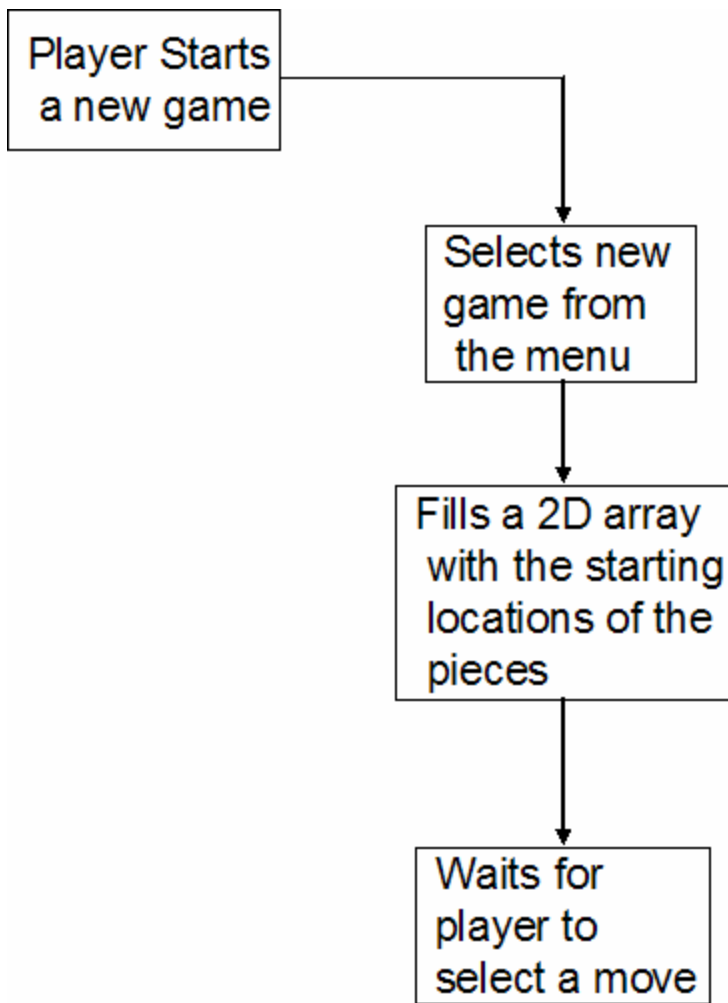


Figure 10: Starting a game

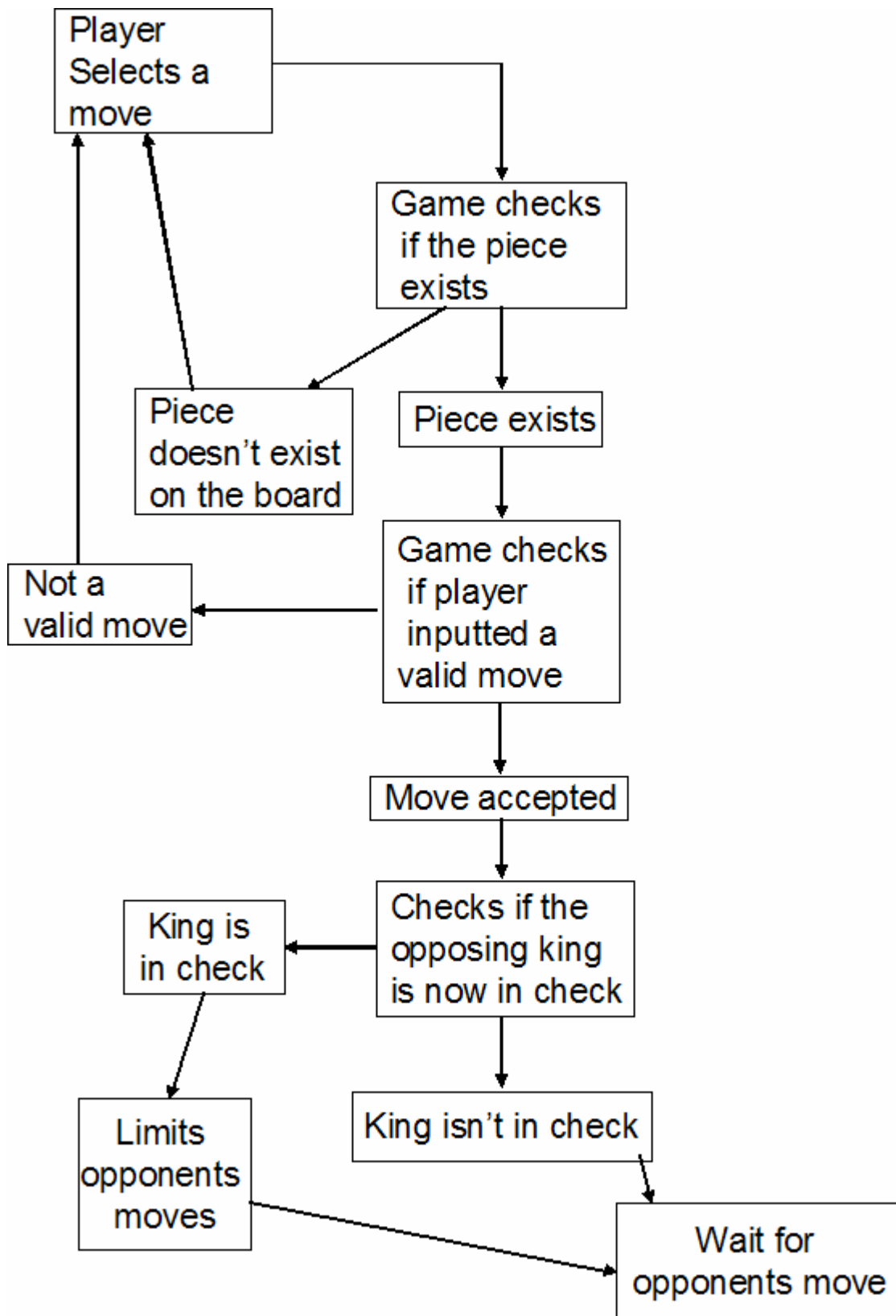


Figure 11: Making a move

A2: Criteria for Success

Proposed Solution Outline

- 1) Allows users to create a profile with the game to keep track of statistics such as wins and losses.
- 2) Games can be started, with or without a profile being used, that follow the general rules of chess.
- 3) Games can be saved at anytime by the players regardless of whether or not the players are using profiles.
- 4) Players can move back and forward through moves they have made note: if they have moved back to a point and changed their move then all moves ahead of that one are gone.

Intended Benefits

Allow users to play with each other as a computer with a basic java compiler is available which almost every computer has.

Keeps track of each individual player's statistics enabling them, and other players, to see what pieces they are more likely to sacrifice in a match.

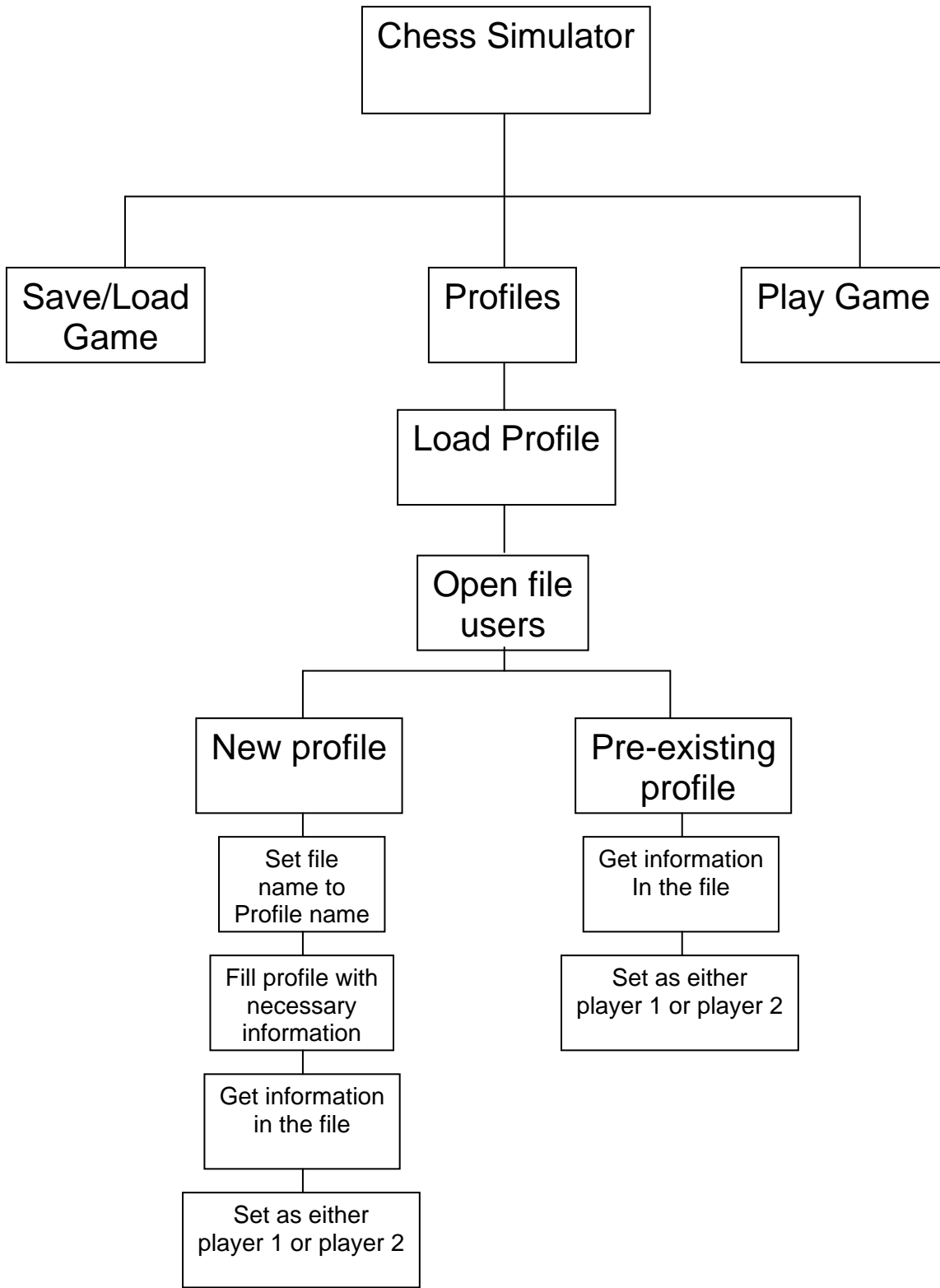
Restrictions/Requirements

The software must be able to run on a standard PC with Java 6.0 installed on it. Expected OS of the computer would be either Windows 2000 or Windows XP however, as long as the computer is able to run Java 6.0 it should be able to run the program.

Goals

- Players will be able to create their own profiles
- Games can be started and saved at any point and started again at anytime
- The game will auto-update the board after every move
- Players can go to pasted moves if they so choose
- Players can check their own statistics as long as the profile is loaded and its their turn
- Players can load their profile at anytime

A3: Prototype Solution



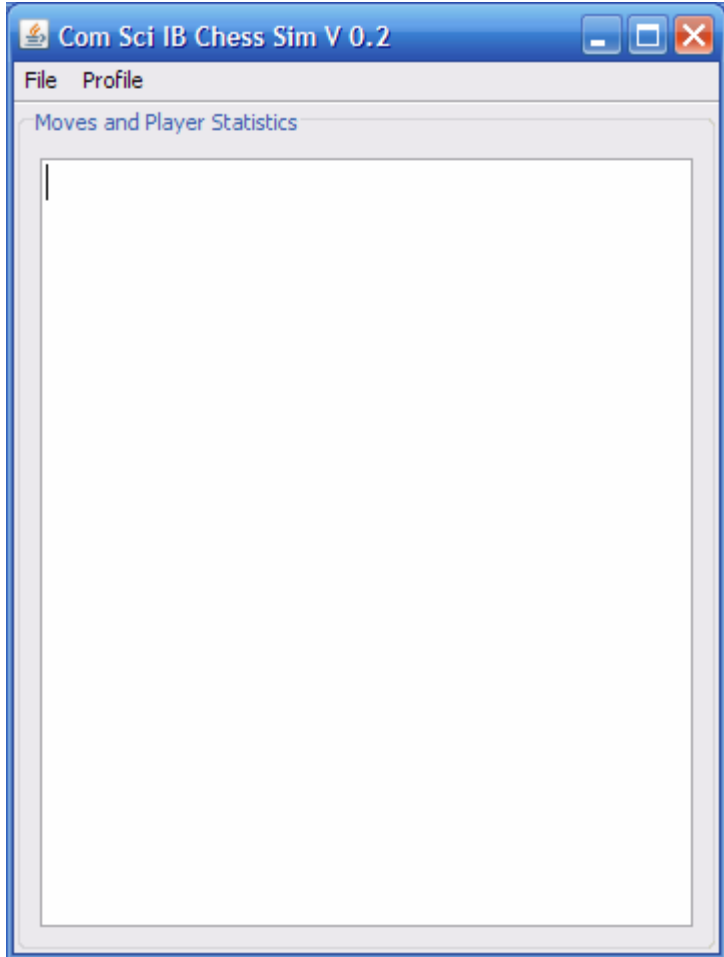


Figure 12: The initial programs appearance

Above is what the user would see upon accessing the prototype giving as few options to the user makes it less likely for them to cause an error.

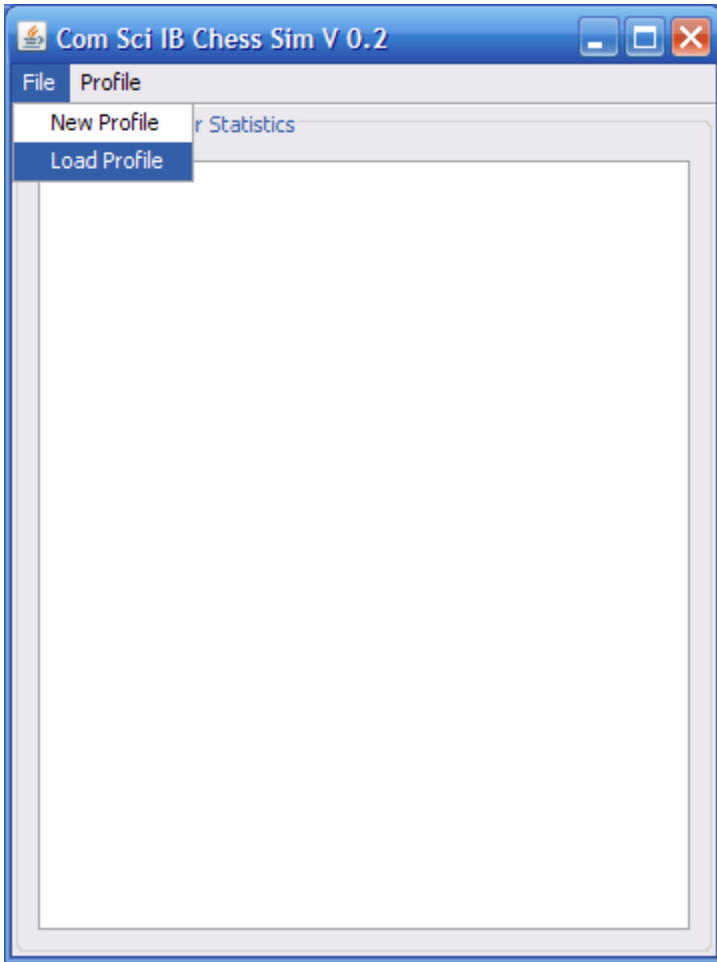


Figure 13: The initial file menu

In the final copy there will be more options in the file menu, but for the purposes of this prototype, only the necessary options will be available.

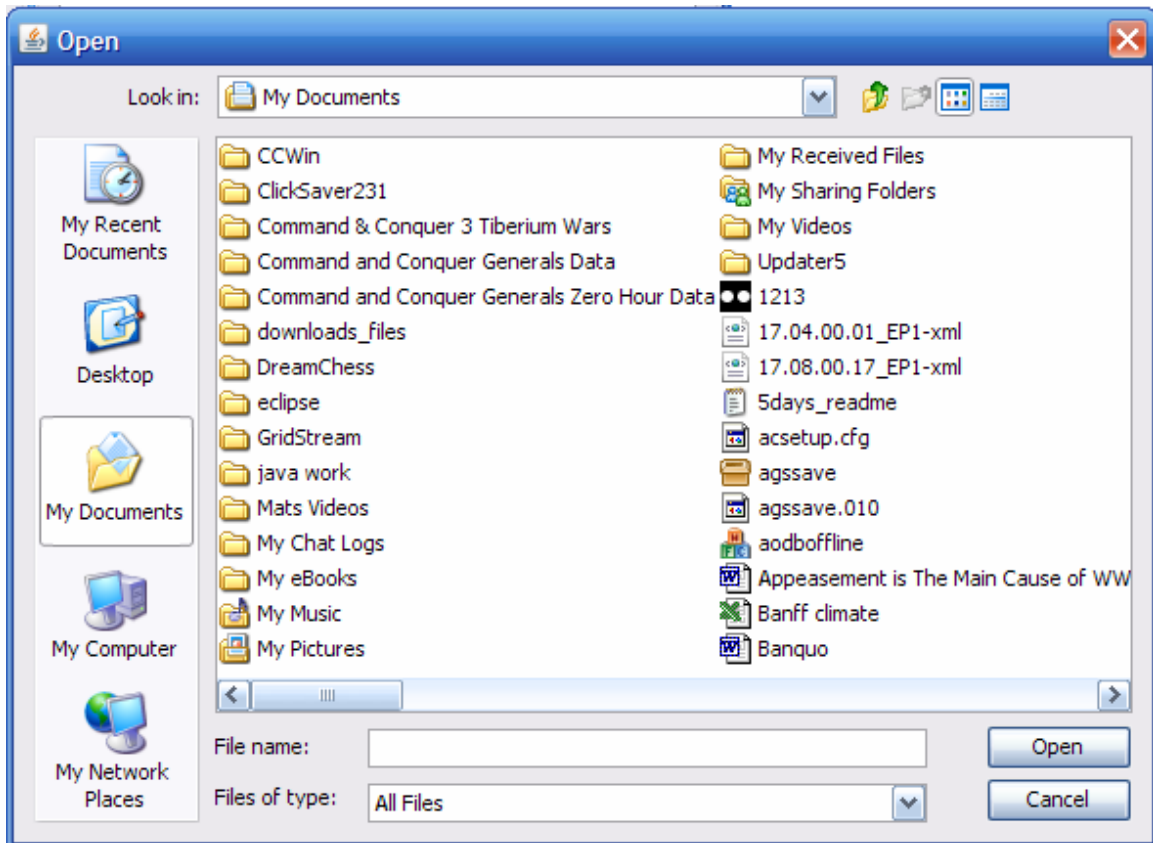


Figure 14: JFileChooser for the user

The above is the File Chooser that is shown to the user. An obvious improvement would be to implement a file filter so the user does not have to look through all of the above files to find the one he is looking for.

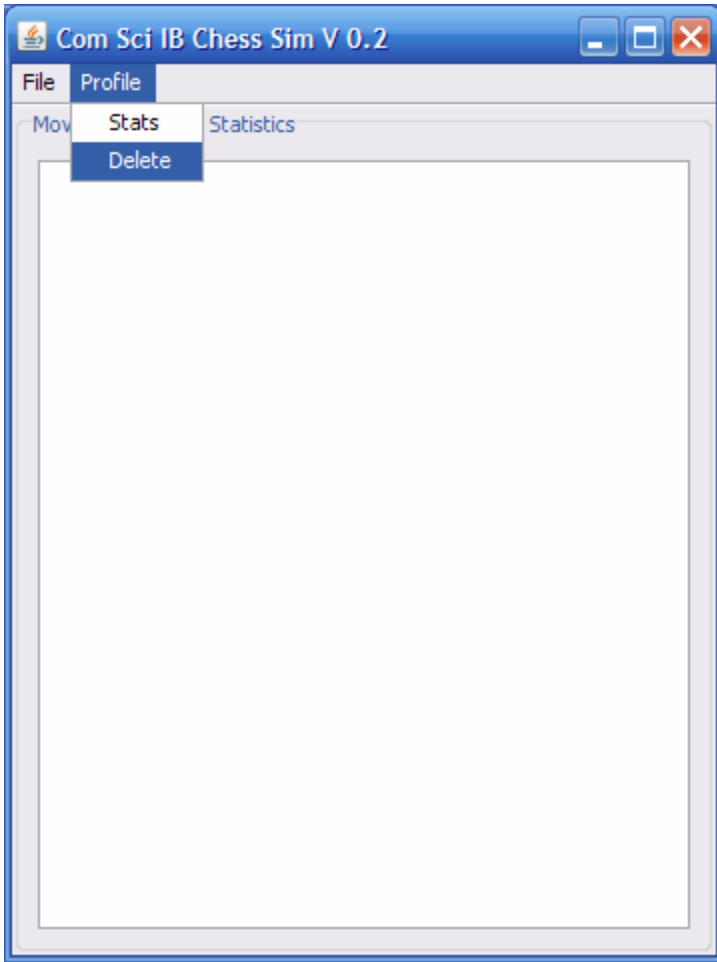


Figure 15: The profile menu

Like the File menu only the necessary options are shown.

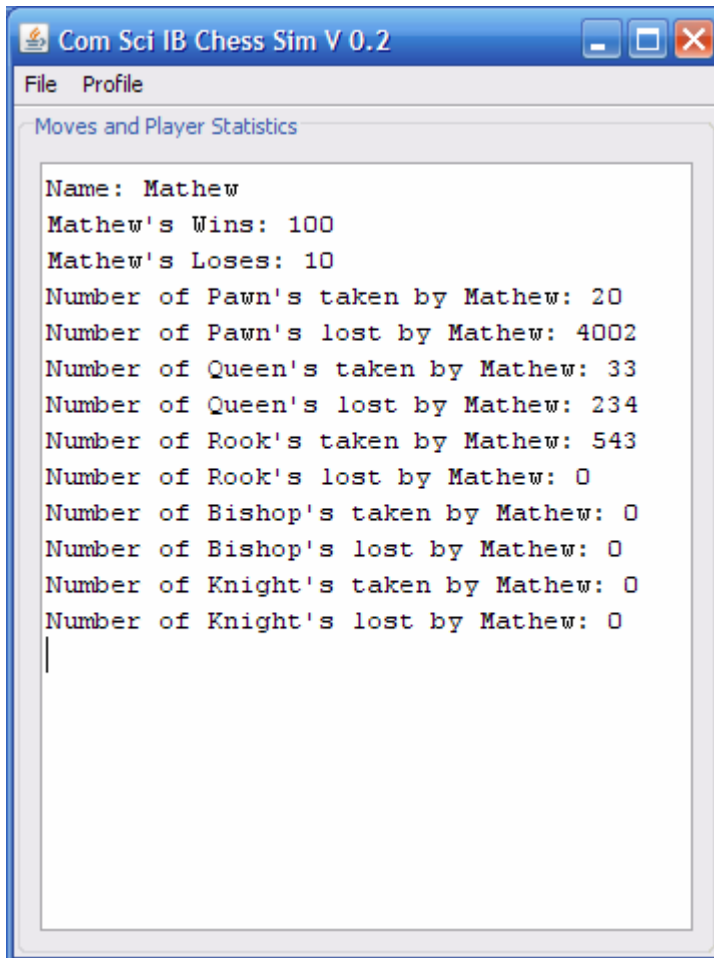


Figure 16: Stats output

The above is the expected output if the user were to select the “Stats” option under profile.

Now that the basic visuals for the prototype have been shown, I will now include the code used to get the above visuals.

```
001 package src;
002 import java.awt.*;
003 import java.awt.event.ActionEvent;
004 import java.awt.event.ActionListener;
005 import java.util.*;
006
007 import javax.swing.*;
008 import java.io.*;
009
010
011
```

```

012 public class Chessboard implements ActionListener
013 {
014     JButton move;// Button to set a move into action once a player has
015         // selected the move parameters.
016     JFrame frame;// Sets the frame so that error message can be displayed
017     JMenuBar bar;// Sets the JMenuBar to be used in more than 1 place
018     JMenu file, profile; // Sets the JMenu's for use in the JMenuBar
019     JTextField input; // The JTextField for players to select the piece and
020         // location they wish to move.
021     JMenuItem Nprofile,Lprofile,delete,stats;// windows JMenuBar.
022     final JFileChooser load = new JFileChooser();// JFileChooser for
023         // loading profiles.
024     final JFileChooser save = new JFileChooser();// JFileChooser for
025         // saving profiles.
026     String player1,player2,newprofilename;// Stores, in order, player 1's
027         // name, player 2's name and the
028         // name of a new profile.
029     RandomAccessFile in, in2, in3;// The I/O streams for Player1, Player2,
030         // and loading games respectively.
031     PrintWriter out;// Used to save games already in progress.
032     JTextArea movesnStats, chessboard;// The output boards for Moves and
033         // Statistic's and Chessboards.
034     JScrollPane scrollMovenStats, scrollChessboard;// Sets the ability to
035         // scroll up and down to MovesnStats
036         // and Chessboard.
037     int currentPlayer=1;// Stores which players turn it is currently 1 for
038         // Player1 0 for Player2.
039     int player1stats[]=new int[12], player2stats[] = new int[12];//total
040         // number of stats each player has
041
042
043     /**
044     * Creates the JMenuBar for the main window.
045     * @return the JMenuBar for the main window.
046     */
047     public JMenuBar createJMenuBar()
048     {
049         bar = new JMenuBar();
050         file = new JMenu("File");
051         Nprofile = new JMenuItem("New Profile");
052         Nprofile.addActionListener(this);
053         file.add(Nprofile);
054         Lprofile = new JMenuItem("Load Profile");
055         Lprofile.addActionListener(this);
056         file.add(Lprofile);
057         bar.add(file);
058         profile = new JMenu("Profile");
059         stats = new JMenuItem("Stats");
060         stats.addActionListener(this);
061         profile.add(stats);
062         delete = new JMenuItem("Delete");
063         delete.addActionListener(this);
064         profile.add(delete);
065         bar.add(profile);
066         return bar;
067     }
068     /**
069     * creates the JTextArea for statisitc's and move output.
070     * @return text output window.
071     */
072     public JScrollPane createMovenStats()
073     {
074         movesnStats = new JTextArea(21,40);// Fits approximatly 1 full

```

```

075             // statistic's output.
076     scrollMovenStats = new JScrollPane(movesnStats);
077     return scrollMovenStats;
078 }
079
080 /**
081  * Sets the JFrame for the program and places the elements in it.
082  */
083 public static void createAndShowGUI()
084 {
085     Chessboard fram = new Chessboard();
086     JFrame frame = new JFrame("Com Sci IB Chess Sim V 0.2");
087     JPanel contentPane = new JPanel(new GridBagLayout());
088     JPanel outputPanel = new JPanel(new GridBagLayout());
089     GridBagConstraints c = new GridBagConstraints();
090     frame.setJMenuBar(fram.createJMenuBar());
091     frame.setContentPane(contentPane);
092     outputPanel.add(fram.createMovenStats(),c);
093     outputPanel.setBorder(BorderFactory.createCompoundBorder
094         (BorderFactory.createTitledBorder("Moves and Player Statistics"),
095         BorderFactory.createEmptyBorder(5,5,5,5)));
096         // Gives a fair amount of space between the boarder
097         // and the TextArea.
098     contentPane.add(outputPanel,c);
099     frame.setVisible(true);
100     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
101     frame.pack();
102 }
103
104 public static void main(String[] args) throws IOException
105 {
106     try{
107         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
108     }
109     catch(UnsupportedLookAndFeelException e){}
110     catch(ClassNotFoundException e){}
111     catch(InstantiationException e){}
112     catch(IllegalAccessException e){}
113     SwingUtilities.invokeLater(new Runnable() {
114         public void run() {
115             createAndShowGUI();}});
116 }
117 /**
118  * The code for events such as button presses and
119  * activation of JMenuItem's.
120  */
121 public void actionPerformed(ActionEvent e)// The event being preformed
122         // IE pushing a button.
123 {
124     Chessboard action = new Chessboard();
125     if(e.getSource()== Lprofile)
126     {
127         try{
128             if(player1==null)
129             {
130                 Object playerinput[] = new Object[13];// 13 is the top number of
131                 // statistic's a player
132                 // can have.
133                 playerinput = action.Loadprofile(null);
134                 if(playerinput[0]!=null)
135                 {
136                     player1 = String.valueOf(playerinput[0]);
137                     for(int i = 1; i<13;i++)// 13 is the top number of

```

```

138             // statistic's a player can have.
139         {
140             player1stats[i-1]=Integer.parseInt(
141                 String.valueOf(playerinput[i]));
142         }
143     }
144 }
145 else if(player2==null)
146 {
147     Object playerinput[] = new Object[13]; // 13 is the top number of
148         // statistic's a player
149         // can have.
150     playerinput = action.Loadprofile(null);
151     if(playerinput[0]!=null)
152     {
153         player2 = String.valueOf(playerinput[0]);
154         for(int i = 1; i<13;i++) // 13 is the top number of
155             // statistic's a player can have.
156         {
157             player2stats[i-1]=Integer.parseInt(
158                 String.valueOf(playerinput[i]));
159         }
160     }
161 }
162 }
163 catch(FileNotFoundException f){}
164 catch(IOException f){}
165 }
166 if(e.getSource()== stats)
167 {
168     if(currentPlayer==1) // currentPlayer is equal to 1 when its player
169         // 1's turn.
170     {
171         if(player1!=null)
172         {
173             movesnStats.append("Name: " + player1 + "\n");
174             movesnStats.append(player1 + "'s Wins: " + String.valueOf(
175                 player1stats[0]) + "\n");
176             movesnStats.append(player1 + "'s Loses: " + String.valueOf(
177                 player1stats[1]) + "\n");
178             movesnStats.append("Number of Pawn's taken by " + player1 +
179                 ": " + String.valueOf(player1stats[2]) + "\n");
180             movesnStats.append("Number of Pawn's lost by " + player1 +
181                 ": " + String.valueOf(player1stats[3]) + "\n");
182             movesnStats.append("Number of Queen's taken by " + player1 +
183                 ": " + String.valueOf(player1stats[4]) + "\n");
184             movesnStats.append("Number of Queen's lost by " + player1 +
185                 ": " + String.valueOf(player1stats[5]) + "\n");
186             movesnStats.append("Number of Rook's taken by " + player1 +
187                 ": " + String.valueOf(player1stats[6]) + "\n");
188             movesnStats.append("Number of Rook's lost by " + player1 +
189                 ": " + String.valueOf(player1stats[7]) + "\n");
190             movesnStats.append("Number of Bishop's taken by " + player1
191                 + ": " + String.valueOf(player1stats[8]) + "\n");
192             movesnStats.append("Number of Bishop's lost by " + player1 +
193                 ": " + String.valueOf(player1stats[9]) + "\n");
194             movesnStats.append("Number of Knight's taken by " + player1
195                 + ": " + String.valueOf(player1stats[10]) + "\n");
196             movesnStats.append("Number of Knight's lost by " + player1 +
197                 ": " + String.valueOf(player1stats[11]) + "\n");
198         }
199     }
200     if(currentPlayer==0) // currentPlayer is equal to 0 when its player

```

```

201         // 2's turn.
202     {
203         if(player2!=null)
204         {
205             movesnStats.append("Name: " + player2 + "\n");
206             movesnStats.append(player2 + "'s Wins: " + String.valueOf(
207                 player2stats[0]) + "\n");
208             movesnStats.append(player2 + "'s Loses: " + String.valueOf(
209                 player2stats[1]) + "\n");
210             movesnStats.append("Number of Pawn's taken by " + player2 +
211                 ": " + player2stats[2] + "\n");
212             movesnStats.append("Number of Pawn's lost by " + player2 +
213                 ": " + player2stats[3] + "\n");
214             movesnStats.append("Number of Queen's taken by " + player2 +
215                 ": " + player2stats[4] + "\n");
216             movesnStats.append("Number of Queen's lost by " + player2 +
217                 ": " + player2stats[5] + "\n");
218             movesnStats.append("Number of Rook's taken by " + player2 +
219                 ": " + player2stats[6] + "\n");
220             movesnStats.append("Number of Rook's lost by " + player2 +
221                 ": " + player2stats[7] + "\n");
222             movesnStats.append("Number of Bishop's taken by " + player2
223                 + ": " + player2stats[8] + "\n");
224             movesnStats.append("Number of Bishop's lost by " + player2 +
225                 ": " + player2stats[9] + "\n");
226             movesnStats.append("Number of Knight's taken by " + player2
227                 + ": " + player2stats[10] + "\n");
228             movesnStats.append("Number of Knight's lost by " + player2 +
229                 ": " + player2stats[11] + "\n");
230         }
231     }
232 }
233 if(e.getSource()== Nprofile)
234 {
235     try{
236         if(player1==null)
237         {
238             Object playerinput[] = new Object[13];
239             playerinput = action.Newprofile();
240             if(playerinput[0]!=null)
241             {
242                 player1 = String.valueOf(playerinput[0]);
243                 for(int i = 1; i<13;i++)
244                 {
245                     player1stats[i-1]=Integer.parseInt(
246                         String.valueOf(playerinput[i]));
247                 }
248             }
249         }
250         else if(player2==null)
251         {
252             Object playerinput[] = new Object[14];
253             playerinput = action.Newprofile();
254             if(playerinput[0]!=null)
255             {
256                 player2 = String.valueOf(playerinput[0]);
257                 for(int i = 1; i<13;i++)
258                 {
259                     player2stats[i-1]=Integer.parseInt(
260                         String.valueOf(playerinput[i]));
261                 }
262             }
263         }

```

```

264     }
265     catch(FileNotFoundException f){}
266     catch(IOException f){}
267 }
268 if(e.getSource() == delete)
269 {
270     if(currentPlayer == 1)
271     {
272         try{
273             in = new RandomAccessFile(player1+".pro", "rw");
274             player1=null;
275             in.seek(0);
276             in.writeBytes("0");
277         }
278         catch(IOException f){}
279     }
280     else if(currentPlayer == 0)
281     {
282         try{
283             in2 = new RandomAccessFile(player2+".pro", "rw");
284             in2.seek(0);
285             in2.writeBytes("0");
286             player2=null;
287         }
288         catch(IOException f){}
289     }
290 }
291 }
292 }
293 /**
294  * Allows the user to load a premade profile with all of their
295  * individual statistic's.
296  * @return object array containing player 1's or player 2's information.
297  * @throws FileNotFoundException
298  * @throws IOException
299  */
300 public Object[] Loadprofile(String newprofile // If a new profile is
301                             // being loaded.
302                             ) throws FileNotFoundException, IOException
303 {
304     Object firstplayer[] = new Object[13]; // Name+all statistics
305                                     // totaling 13
306     Object secondplayer[] = new Object[13]; // Name+all statistics
307                                     // totaling 13
308     Object outputarray[] = new Object[13]; // Will contain 1 of the two
309                                     // above to simplify output
310     newprofilename = newprofile;
311     if(newprofilename != null)
312     {
313         if(player1 == null)
314         {
315             player1 = newprofilename;
316             try{
317                 in = new RandomAccessFile(player1+".pro", "r");
318             }
319             catch(FileNotFoundException e)
320             {
321                 JOptionPane.showMessageDialog(
322                     frame, "FileNotFoundException: " + e.getMessage());
323             }
324             StringTokenizer st = new StringTokenizer(
325                 in.readLine().replaceAll("_", " "));
326             st.nextToken(); //jumps the deleted token
327             st.nextToken(); //jumps the name token
328             firstplayer[0] = player1;

```

```

327         for(int i = 1;i<13;i++)
328         {
329             firstplayer[i]=st.nextToken();
330         }
331
332         outputarray = firstplayer;
333     }
334 }
335 if(player2==null)
336 {
337     player2=newprofilename;
338     try{
339         in2 = new RandomAccessFile(player2+".pro", "r");
340     }
341     catch(FileNotFoundException e)
342     { JOptionPane.showMessageDialog(
343         frame, "FileNotFoundException: " + e.getMessage());}
344     StringTokenizer st = new StringTokenizer(
345         in2.readLine().replaceAll("_", " "));
346     st.nextToken();//jumps the deleted token
347     st.nextToken(); //jumps the name token
348     secondplayer[0]=player2;
349     for(int i = 1;i<13;i++)
350     {
351         secondplayer[i]=st.nextToken();
352     }
353     outputarray = secondplayer;
354 }
355 }
356 }
357 }
358 if(newprofilename==null)
359 {
360     if(player1==null)
361     {
362         int returnVal = load.showOpenDialog(frame);
363         if(returnVal == JFileChooser.APPROVE_OPTION) {
364             try{
365                 StringTokenizer inst =
366                 new StringTokenizer(
367                 load.getSelectedFile()
368                 .getName().replace(".", " "));
369                 inst.nextToken();//Removes the files, name extension
370                 // is all that is needed right now.
371                 String fileExtension = inst.nextToken();
372                 if(fileExtension.equals("pro"))
373                 {
374                     in = new RandomAccessFile(
375                     load.getSelectedFile().getName(),"r");
376                     StringTokenizer st =
377                     new StringTokenizer(
378                     in.readLine()
379                     .replaceAll("_", " ").trim());
380                     if(st.nextToken().equals("0"))
381                     {
382                         JOptionPane.showMessageDialog(
383                         frame, "Profile has been deleted.");
384                         in.close();
385                     }
386                     else
387                     {
388                         firstplayer[0] = st.nextToken();
389                         for(int i=1;i<13;i++)

```

```

390         {
391             firstplayer[i]=st.nextToken();
392         }
393     }
394 }
395     if(!fileExtention.equals("pro"))
396     {JOptionPane.showMessageDialog(
397         frame, "File Selected is not a profile.");}
398 }
399     catch(FileNotFoundException e)
400     { JOptionPane.showMessageDialog(frame,
401         "FileNotFoundException: " + e.getMessage());}
402     catch(IOException e)
403     { JOptionPane.showMessageDialog(
404         frame, "IOException: " + e.getMessage());}
405 }
406 outputarray = firstplayer;
407 }
408 else
409 {
410     int returnVal = load.showOpenDialog(frame);
411     if(returnVal == JFileChooser.APPROVE_OPTION) {
412         try{
413             if(load.getSelectedFile().getName()!=null)
414             {
415                 StringTokenizer inst =
416                     new StringTokenizer(
417                         load.getSelectedFile()
418                             .getName().replace(".", " "));
419                 inst.nextToken();// Removes the files name extention
420                     // is all that is needed right now.
421                 String fileExtention = inst.nextToken();
422                 if(fileExtention.equals("pro"))
423                 {
424                     in2 = new RandomAccessFile(
425                         load.getSelectedFile().getName(),"r");
426                     StringTokenizer st =
427                         new StringTokenizer(
428                             in2.readLine()
429                                 .replaceAll("_", " ").trim());
430                     if(st.nextToken().equals("0"))
431                     {
432                         JOptionPane.showMessageDialog(
433                             frame, "Profile has been deleted.");
434                         in2.close();
435                     }
436                     else
437                     {
438                         secondplayer[0] = st.nextToken();
439                         for(int i=1;i<13;i++)
440                         {
441                             secondplayer[i]=st.nextToken();
442                         }
443                     }
444                 }
445                 if(!fileExtention.equals("pro"))
446                 {JOptionPane.showMessageDialog(
447                     frame, "File Selected is not a profile.");}
448             }
449         }
450         catch(FileNotFoundException e)
451         { JOptionPane.showMessageDialog(
452             frame, "FileNotFoundException: " + e.getMessage());}

```

```

453         catch(IOException e)
454         { JOptionPane.showMessageDialog(
455             frame, "FileNotFoundException: " + e.getMessage());}
456         }
457         outputarray = secondplayer;
458     }
459 }
460 return outputarray;
461 }
462 /**
463  * Updates the players profile after each game or when a game is saved.
464  * @throws IOException
465  */
466 public void Saveprofile() throws IOException
467 {
468     if(player1!=null)
469     {
470         in = new RandomAccessFile(player1+".pro","rw");
471         in.seek(27);//skips the players name.
472         for(int i=0;i<13;i++)
473         {
474             int maxint = 5;// Used to standardize the length of output
475                 // numbers.
476             String player1stat = String.valueOf(player1stats[i]);
477             while(player1stat.length()<=maxint)
478                 {player1stat = player1stat+"_";}
479             in.writeBytes(player1stat);
480         }
481     }
482     if(player2!=null)
483     {
484         in = new RandomAccessFile(player2+".pro","rw");
485         in.seek(27);//skips the players name.
486         for(int i=0;i<13;i++)
487         {
488             int maxint = 5;
489             String player2stat = String.valueOf(player2stats[i]);
490             while(player2stat.length()<=maxint)
491                 {player2stat = player2stat+"_";}
492             in.writeBytes(player2stat);
493         }
494     }
495 }
496 /**
497  * Sets up a players new profile and loads it into either player1
498  * or player2 using Loadprofile.
499  * @throws IOException
500  * @retrun the new player information
501  */
502 public Object[] Newprofile() throws IOException
503 {
504     Object playerinput[] = new Object[14];
505     Chessboard newplayer = new Chessboard();
506     save.setDialogType(JFileChooser.OPEN_DIALOG);
507     int returnVal = save.showOpenDialog(frame);
508     if(returnVal == JFileChooser.APPROVE_OPTION)
509     {
510         try{
511             String inputname = save.getName(save.getSelectedFile());
512             if(inputname!=null)
513             {
514                 in = new RandomAccessFile(inputname+".pro","rw");
515                 newprofilename = "1_" + save.getName(save.getSelectedFile())

```

```

516         .replaceAll(".pro", " ").trim();
517     int count = newprofilename.length();
518     while(count<27){newprofilename=newprofilename+"_";count++;}
519     in.setLength(87);// 27 for name, 6 for colour, and 5 for
520         // each score catagory.
521     in.writeBytes(newprofilename);
522     for(int i=0;i<12;i++)
523     {
524         in.writeBytes("0____");
525     }
526
527     if(player1==null)
528     {
529         playerinput = newplayer.Loadprofile(inputname);
530     }
531     else if(player2==null)
532     {
533         playerinput = newplayer.Loadprofile(inputname);
534     }
535     }
536     }
537     catch(IOException e){JOptionPane.showMessageDialog(
538         frame, "IOException: " + e.getMessage());}
539 }
540 return playerinput;
541 }
542 }
543 }

```

B1: Data Structure

Arrays

Since the storage of multiple groups of numbers in a predefined order is going to be part of the profiles, an array makes it easy to keep track of where everything associate with the player, excluding their name, is kept.

The array will be updated each time a piece is taken or a game is won/lost so it will stay up-to-date with the information it contains. The array is probably only going to contain numbers and an output of it would contain a statement that has already been pre-determined such as “ ‘Player 1’ has taken 1003 Pawns.” Where the 1003 is the number

that would be stored in the array and Player 1 means the player who has taken those pawns, Player 1 will be stored elsewhere to avoid confusion.

Two Dimensional Arrays

This second type of array would be the most logical choice for saving the locations of the pieces on the board while the program is running. Since a 2D array has both an x and a y coordinate to them, just like the board, it can easily be used for behind the scene's movement of pieces only being outputted when a move is successful.

It would likely contain the Piece on that space and the player that owns that piece since little else in the way of information is necessary to be kept there. The format would probably be something like "P1" for player 1 or "P2" for player 2 and piece names such as "K1" relating to the knight on the left hand side of the board. Which brings up a point, there must be something to distinguish the different pieces from one another so it is likely that each piece will be titled with a 1, 2... depending on where the piece is on the board. An example of what one of player 1's pawns might be saved as would be "P1 P2" this would be known as player 1's second pawn, from the left.

Data Files

Data files will contain two different types of data. The first type will contain the profiles and use the extension .pro (since it's the first three letters of profile). The information contained within will first be the data on whether or not the profile counts as

deleted. If there is a 1, it will count as existing. If there is a 0m it will count as deleted and the program will immediately close the file. The next will be the players name, it will contain anywhere from 1-25 characters and match the name of the file to make it easier for the user to find. Each of them will be standardized so that any empty space afterwards will be turns into “_”. After this, there will be the series of 12 numbers, each standardized to a length of 5, until the end of the file so the data in the file will look something like

1_Mathew_____100__10__20__4002_33__234__543__0__0__
 _0__0__0__.

The alternative type of data will be that of a saved game using the extension .gam (the first 3 letters in game). The first piece of data in the file will be whose turn it was when the game is saved, 1 for player 1 and 0 for player 2. Following this will be the coordinates of a piece, whose piece it is and finally what piece it is. The last part will be repeated until the end of the file with a maximum of 32 since there can be no more than 32 piece out at 1 time. 0 0 0 P1 R1 0 1 P1 K1 0 2 P1 B1 0 3 P1 Qu 0 4 P1 Ki 0 5 P1 B2 0 6 P1 K2 0 7 P1 R2 1 0 P1 P1 1 1 P1 P2 1 2 P1 P3 1 4 P1 P5 1 5 P1 P6 1 6 P1 P7 1 7 P1 P8 3 3 P1 P4 6 0 P2 P1 6 1 P2 P2 6 2 P2 P3 6 3 P2 P4 6 4 P2 P5 6 5 P2 P6 6 6 P2 P7 6 7 P2 P8 7 0 P2 R1 7 1 P2 K1 7 2 P2 B1 7 3 P2 Qu 7 4 P2 Ki 7 5 P2 B2 7 6 P2 K2 7 7 P2 R2 that is a sample output as u can clearly see, it was player 2’s turn when it was saved and Player 1 still had a rook in the upper left corner of the map (this is shown as the “0 0 P1 R1”).

Stacks

The final major data structure is the stack. This will be implemented to allow the player(s) to move back and forth through the match. They will be split into a forward stack and a backward stack. The forward stack is going to be considerably simpler than the backward stack since it does not have to keep track of piece that might have been lost. The forward stack will contain the coordinates the piece was at during that move, whose piece it was and what piece it was. It would probably contain something like "P1 P3 3, 3" signifying that Player 1's third pawn was at 3, 3 during this turn.

The backwards stack will need to have an extra bit on the end from the forward stack. Since piece need to be replaced on the field if they were taken, they also need to be stored. However, unlike with the piece that was moved, the coordinates of the taken piece do not necessarily need to be saved. Since we already know where the taken piece was, since in the same location as the piece that was moved is now, we can just replace them so an entry in the stack would likely look like "P1 P3 3, 4 P2 K1" The first bit is the same, where it differs is in the end that states that player 2's first knight was taken if no piece was taken then it would instead have "P N" where the "P2 K1" is.

The stack works by adding a new element to the end of an array. If there is no more space in the array, then a private method would be called to increase the size of the stack adding more space for the element to occupy. Then the program can either take the top element from the stack and output it, deleting it from the stack in the process, or just output the top element without removing it so the stack stays intact but what is contained within is still known.

B2: Algorithms

Creating the JMenuBar:

```
/**
 * Creates the JMenuBar for the main window.
 * @return the JMenuBar for the main window.
 * @pre the program has started and the CreateAndShowGUI
 * method has been called
 * @post the JMenuBar has been created and bundled up ready to be
 * implemented in the GUI
 */
```

Creating the moveButton:

```
/**
 * creates the JButton to activate a move.
 * @return a JButton.
 * @pre the program has started and the CreateAndShowGUI
 * method has been called
 * @post the moveButtons ActionListener has been created and
 * bundled up ready to be implemented in the GUI
 */
```

Creating the JScrollPane for the movesAndStatistics:

```
/**
 * creates the JTextArea for statistics and move output
 * and adds it to a JScrollPane.
 * @return text output window with scroll bar.
 * @pre the program has started and the CreateAndShowGUI
 * method has been called
```

```
* @post the movesandStatistics JTextArea has been created and
* bundled up ready to be implemented in the GUI
*/
```

Creating the JScrollPane for the chessboard:

```
/**
* creates the JTextArea for chessboard output and adds
* it to a JScrollPane.
* @return text output window with scroll bar.
* @pre the program has started and the CreateAndShowGUI
* method has been called
* @post the chessboard JTextArea has been created and
* bundled up ready to be implemented in the GUI
*/
```

Creating the JTextField for playerinputs:

```
/**
* Player input field for them to select a move.
* @return TextField for player to input moves.
* @pre the program has started and the CreateAndShowGUI
* method has been called
* @post the TextField for player input moves has been created
* and bundled up ready to be implemented in the GUI
*/
```

Creating the overall GUI interface:

```
/**
* Sets the JFrame for the program and places the elements in it.
* @pre the program has started
* @post the GUI will be build and standing by for users to
* activate other methods
```

```
*/
```

Creating the ActionListener:

```
/**  
 * The code for events such as button presses and  
 * activation of JMenuItem.  
 * @pre the user has selected something i.e. they have selected  
 * new profile  
 * @post the selected option has been completed and the data is  
 * updated  
 */
```

Creating the visual representation of the board:

```
/**  
 * Outputs the board to the Chessboard for so the player can see  
 * it.  
 * @param board the board that you want to output  
 * @param movesnStats the JTextArea being outputed to.  
 * @pre a game must already be started either from scratch or a  
 * saved game  
 * @post the chessboard Text Area will have a new chessboard on  
 * it  
 */
```

Creating the Save game option:

```
/**  
 * Outputs the game to a file that can be loaded later.  
 * @param savegame the game u want to save  
 * @throws IOException  
 * @pre a game must already be started either from scratch or a  
 * saved game
```

```
* @post there will be a new .gam file in the project folder
*/
```

Creating a new game:

```
/**
 * Sets up a new game.
 * @return newGame An array containing the pieces starting
 * locations.
 * @pre The GUI must be built
 * @post Chessboard will be visible and everything ready to play
 * a game
 */
```

Creating load game option:

```
/**
 * allows players to load previously saved games.
 * @return the previously saved game.
 * @throws FileNotFoundException
 * @pre The GUI must be built
 * @post Chessboard will be visible and everything ready to play
 * a game
 */
```

Creating load profile option:

```
/**
 * Allows the user to load a premade profile with all of their
 * individual statistic's.
 * @return object array containing player 1's or player 2's
 * information.
 * @throws FileNotFoundException
 * @throws IOException
```

```
* @pre The GUI must be built
* @post A player profile will be loaded ready to output and
* update statistics
*/
```

Creating the save profile option:

```
/**
 * Updates the players profile after each game or when a game is
 * saved.
 * @throws IOException
 * @pre A game must already have been played/must be being played
 * @post A player profile will be updated
 */
```

Creating the new profile option:

```
/**
 * Sets up a players new profile and loads it into either player1
 * or player2 using Loadprofile.
 * @throws IOException
 * @return the new player information
 * @pre The GUI must be built and the new profile option selected
 * @post A new profile will have been made and loaded
 */
```

Creating Chess moves constructor:

```
/**
 * Constructor of the program that builds a new board that can
 * be updated and allows for the player to move pieces.
 * @pre The GUI must be built and a new game must be started
 * @post The board is ready to be updated
 */
```

Creating the update for the chess moves constructor:

```
/**
 * Updates the information of the board by inputing a new board.
 * @param inputboard contains the board
 * @pre Chess moves constructor must already have been called
 * and new game must have been selected from the GUI
 * @post The game is ready to be played
 */
```

Creating outputting the board:

```
/**
 * Outputs the board containing the pieces locations.
 * @return board
 * @pre The GUI must be built and a new game must be started
 * @post The board will be sent to where ever it was called
 */
```

Creating the move piece algorithm:

```
/**
 * Checks if the piece being moved exists and if its a legal
 * move.
 * @param peice
 * @param xo
 * @param yo
 * @param xn
 * @param yn
 * @param currentPlayer
 * @return returnMove contains whose turn it now is, where the
 * piece was, where it is now, and if it took a piece.
 * @pre The player must have pressed the move button
 */
```

```
* @post If it is a valid move, the pieces location on the board
* will have changed
*/
```

Creating is there a piece there algorithm:

```
/**
* Checks if a piece is on the specified position
* @param x
* @param y
* @return true if a piece is there, false if there isn't
* @pre A game must have been started or loaded
* @post If there is a piece there it will be true if there isn't
* it will be false
*/
```

Creating the check if the move is in the same column as the piece:

```
/**
* Checks if a piece can move within the column its on.
* @param peice
* @param playermoving
* @param xn
* @param xo
* @param yo
* @return true if it can false if it can't
* @pre A move must have been declared
* @post If it is a valid move, it will return true otherwise
* it will return false
*/
```

Creating the check if the move is in the same row as the piece:

```
/**
 * Checks if the piece can move within the row its on does not
 * include
 * castling.
 * @param peice
 * @param playermoving
 * @param yn
 * @param yo
 * @param xo
 * @return true if it can, false if it can't
 * @pre A move must have been declared
 * @post If it is a valid move, it will return true otherwise
 * it will return false
 */
```

Creating the check if the move is diagonal to the piece:

```
/**
 * Checks if a piece can move diagonally (any of the four
 * directions)
 * based on its location.
 * @param peice
 * @param player
 * @param xo
 * @param xn
 * @param yo
 * @param yn
 * @return true if it can, false if it can't
 * @pre A move must have been declared
```

```
* @post If it is a valid move, it will return true otherwise
* it will return false
*/
```

Creating the check if the move is a possible move for a pawn:

```
/**
 * Checks if the move selected is legal for a Pawn to make
 * includes
 * En passant but not promotion
 * @param peice
 * @param playermoving
 * @param peiceToMove
 * @param yn
 * @param yo
 * @param xn
 * @param xo
 * @return true if it can, false if it can't
 * @pre A move must have been declared
 * @post If it is a valid move, it will return true otherwise
 * it will return false
*/
```

Creating the check if the move is a possible move for a king:

```
/**
 * Checks all of the possible king moves including if he would
 * be in check, does not include castling.
 * @param peice
 * @param playermoving
 * @param yn
 * @param yo
 * @param xn
 * @param xo
 * @return true if he can, false if he can't
 * @pre A move must have been declared
 * @post If it is a valid move, it will return true otherwise
 * it will return false
 */
```

Creating the check if the move is a possible move for a knight:

```
/**
 * Checks all of the possible moves a knight can make.
 * @param peice
 * @param playermoving
 * @param yn
 * @param yo
 * @param xn
 * @param xo
 * @return true if he can move, false if he can't
 * @pre A move must have been declared
 * @post If it is a valid move, it will return true otherwise
 * it will return false
 */
```

```
*/
```

Creating the checking system to tell if a king is in check:

```
/**  
 * Checks if a king would be put into check by the move chosen by  
 * the player.  
 * @param piece  
 * @param playermoving  
 * @param xn  
 * @param yn  
 * @return true if he would be in check, false if he wouldn't  
 * @pre A move must have been declared  
 * @post If the king is in check, it will return true otherwise  
 * it will return false  
*/
```

Creating the checkmate system so a game ends when the king can't move:

```
/**  
 * Checks if the game should end because of checkmate.  
 * @param peice  
 * @param piece  
 * @param playerChecking  
 * @param xKing  
 * @param yKing  
 * @return true if the game will end, false if it will continue.  
 * @pre A move must have been declared  
 * @post If the king is in checkmate, it will return true  
 * otherwise it will return false  
*/
```

Creating the stack constructor:

```
/**
 * Constructs a new stack with space for the first item.
 * @pre The stack is called to be created
 * @post There is a new stack
 */
```

Creating the push algorithm for the stack:

```
/**
 * Adds items to the stack and will increase the size of the
 * stack
 * if it is already full.
 * @param e
 * @pre That a stack exists
 * @post The stack will contain a new element
 */
```

Creating the pop algorithm for the stack:

```
/**
 * Returns the top element on the stack and removes it from the
 * stack.
 * @return returnValue
 * @pre The stack contains at least 1 element
 * @post The element at the top of the stack is outputted and
 * removed from the stack
 */
```

Creating the peek algorithm for the stack:

```
/**
 * Looks at the top value of the stack but does not remove it.
 * @return returnValue
```

```
* @pre The stack contains at least 1 element
* @post The element at the top of the stack is outputted
*/
```

Creating the check for if a stack is empty:

```
/**
 * Checks if the stack does not contain any elements.
 * @return true if there aren't any, false if there are
 * @pre A stack must exist
 * @post true if there are not elements, false otherwise
 */
```

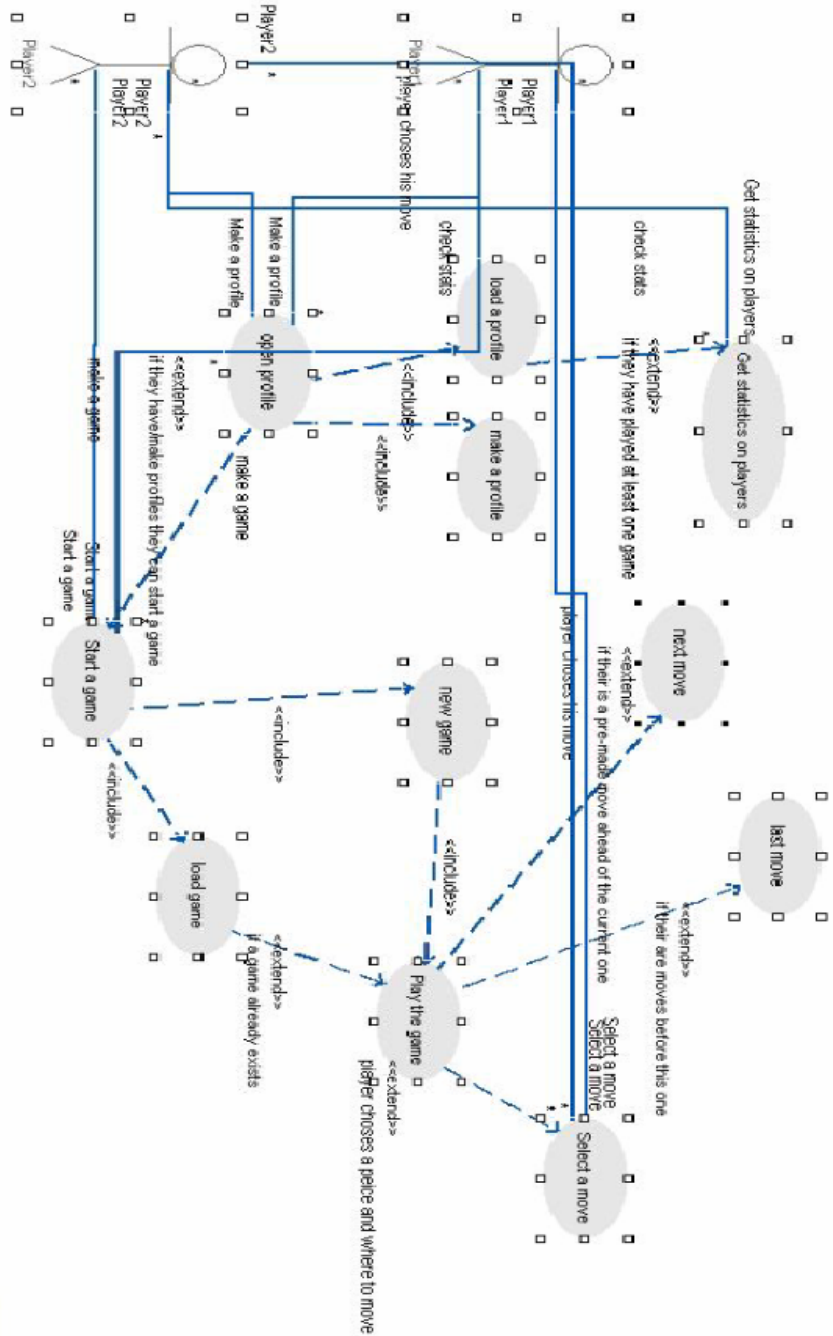
Creating the check for if a stack is full:

```
/**
 * Checks if the stack does not contain any empty space.
 * @return true if there are no null spaces, false if there are
 * @pre A stack must exist
 * @post true if there are no null spaces, false otherwise
 */
```

Creating the algorithm to allow the stack to grow:

```
/**
 * Doubles the space in the stack if it is full and a
 * new element is trying to be added to the top of the stack.
 * @pre stack must be full
 * @post the stack will be twice its original size
 */
```

B3: Modular Organization



UserCaseDiagram srcUseCaseDiagram

Chess Sim User stories

Title: Open a Profile		
AccTest: Openprofile0	Priority: 1	Story Points: 1
When the Chess Sim program is run, both users will load or create their profiles so they can keep track of game information.		

Title: Load a Profile		
AccTest: Lprofile1	Priority: 1	Story Points: 1
The users can load a profile, if they have one, so that they can check their particular statistics or update them though matches.		

Title: New Profile		
AccTest: Nprofile1	Priority: 1	Story Points: 1
By creating a new profile, a user is now able to keep track of their own individual performance and play games.		

Title: Get statistics on player		
AccTest: Stats1	Priority: 2	Story Points: 2
By selecting stats from the menu, the game will place output all of the player's statistics into the text field at the bottom of the window.		

Title: Start a game		
AccTest: Sgame1	Priority: 1	Story Points: 2
With the player profiles loaded, a game can now be started. The user is then given the option to load an old saved game or start a new game.		

Title: New game		
AccTest: Ngame1	Priority: 2	Story Points: 2
Each player is given the option of preset colour's for pieces with black vs. white being the default. A standard match will now proceed with all the rules of chess being placed under the help menu.		

Title: Load Game		
AccTest: Lgame1	Priority: 2	Story Points: 2
If the users elect to continue an old game, all the pre-chosen elements will be loaded from the save file.		

Title: Play the game		
AccTest: Possiblemoves1	Priority: 3	Story Points: 3
With the game now under way, the users will select pieces to move and at any time the users can elect to save the game and come back later.		

Title: Last Move		
AccTest: Lmove1	Priority: 4	Story Points: 3
The current player can move back a maximum of 3 turns so that they can retry a strategy or look over the last few moves if they feel they need to.		

Title: Next Move		
AccTest: Nmove1	Priority: 4	Story Points: 3
If the players have gone back though the moves in the game (maxed at 3), the players can use this option to move the pieces back to where they were as of the last legal move.		

Title: Select a move		
AccTest: Ngame1	Priority: 2	Story Points: 2
Where to move them with the program showing all of the different move choices by changing the background colour of tiles the piece can move to.		

C1: Using good programming style

```
/**=====
* Class:      Chessboard      IB Dossier      Author: Mathew George
* Version:    001              Date:   November 9, 2007
*
* Purpose     Builds the GUI and ActionListener's for a chessboard game
*
* Course:     Computer Science 33 IB          Teacher:  Gerry Donaldson
*
* School:     Sir Winston Churchill High School, Calgary, Alberta, Canada
*
* Language:   Java J2SE 6.0 Target Operating System: Java Virtual Machine
*
* System:     Intel Celeron 3GHz running under Windows XP IDE: Eclipse 3.2
*
=====*/
0001 package src;
0002 import java.awt.*;
0003 import java.awt.event.ActionEvent;
0004 import java.awt.event.ActionListener;
0005 import java.util.*;
0006
0007 import javax.swing.*;
0008 import java.io.*;
0009
0010
0011
0012 public class Chessboard extends ChessMoves implements ActionListener
0013 {
0014     ChessMoves board = new ChessMoves();// Allows the program to call its
0015         // own methods
0016     JButton move;// Button to set a move into action once a player has
0017         // selected the move parameters.
0018     JFrame frame;// Sets the frame so that error message can be displayed
0019     JMenuBar bar;// Sets the JMenuBar to be used in more than 1 place
0020     JMenu file, profile; // Sets the JMenu's for use in the JMenuBar
0021     JTextField input; // The JTextField for players to select the piece and
0022         // location they wish to move.
0023     JMenuItem Ngame,Lgame,Sgame,Lmove, // All the JMenuItem's for the main
0024         Nmove,Nprofile,Lprofile,delete,stats;// windows JMenuBar.
0025     final JFileChooser load = new JFileChooser();// JFileChooser for
0026         // loading profiles.
0027     final JFileChooser save = new JFileChooser();// JFileChooser for
0028         // saving profiles.
0029     String currentGame[][] = new String[8][8];// Used to transfer the board
0030         // around outside of the
0031         // ChessMoves Class.
0032         // Is 8,8 because that is the
0033         // maximum size of a chessboard.
0034         // Any future refence to the number 8
0035         // will be in the same context as here.
0036     String player1,player2,newprofilename;// Stores, in order, player 1's
0037         // name, player 2's name and the
0038         // name of a new profile.
0039     Stack pastedMoves = new Stack();// Stores previous moves made by the
0040         // player.
0041     Stack movesAhead = new Stack();// If pasted moves is popped, it stores
0042         // those values.
0043     RandomAccessFile in, in2, in3;// The I/O streams for Player1, Player2,
```

```

0044         // and loading games respectively.
0045 PrintWriter out;// Used to save games already in progress.
0046 JTextArea movesnStats, chessboard;// The output boards for Moves and
0047         // Statistic's and Chessboards.
0048 JScrollPane scrollMovenStats, scrollChessboard;// Sets the ability to
0049         // scroll up and down to MovesnStats
0050         // and Chessboard.
0051 int currentPlayer=1;// Stores which players turn it is currently 1 for
0052         // Player1 0 for Player2.
0053 int player1stats[]=new int[12], player2stats[] = new int[12];//total
0054         // number of stats each player has
0055
0056
0057
0058 public JMenuBar createJMenuBar()
0059 {
0060     bar = new JMenuBar();
0061     file = new JMenu("File");
0062     Nprofile = new JMenuItem("New Profile");
0063     Nprofile.addActionListener(this);
0064     file.add(Nprofile);
0065     Lprofile = new JMenuItem("Load Profile");
0066     Lprofile.addActionListener(this);
0067     file.add(Lprofile);
0068     file.addSeparator();
0069     Ngame = new JMenuItem("New Game");
0070     Ngame.addActionListener(this);
0071     file.add(Ngame);
0072     Sgame = new JMenuItem("Save Game");
0073     Sgame.addActionListener(this);
0074     file.add(Sgame);
0075     Lgame = new JMenuItem("Load Game");
0076     Lgame.addActionListener(this);
0077     file.add(Lgame);
0078     file.addSeparator();
0079     Lmove = new JMenuItem("Last Move");
0080     Lmove.addActionListener(this);
0081     file.add(Lmove);
0082     Nmove = new JMenuItem("Next Move");
0083     Nmove.addActionListener(this);
0084     file.add(Nmove);
0085     bar.add(file);
0086     profile = new JMenu("Profile");
0087     stats = new JMenuItem("Stats");
0088     stats.addActionListener(this);
0089     profile.add(stats);
0090     delete = new JMenuItem("Delete");
0091     delete.addActionListener(this);
0092     profile.add(delete);
0093     bar.add(profile);
0094     return bar;
0095 }
0096 /**
0097  * Sets up the move button and its actionlistener.
0098  * @return The move button.
0099  */
0100 public JButton moveButton()
0101 {
0102     move = new JButton("Enter move");
0103     move.addActionListener(this);
0104     return move;
0105 }
0106 /**

```

```

0107     * creates the JTextArea for statistc's and move output.
0108     * @return text output window.
0109     */
0110     public JScrollPane createMovesnStats()
0111     {
0112         movesnStats = new JTextArea(21,40);// Fits approximatly 1 full
0113             // statistic's output.
0114         scrollMovenStats = new JScrollPane(movesnStats);
0115         return scrollMovenStats;
0116     }
0117     /**
0118     * creates the JTextArea for the chessboard output.
0119     * @return text output window for boards.
0120     */
0121     public JScrollPane createChessboardText()
0122     {
0123         chessboard = new JTextArea(21,36);// Large enough to fit approximatly
0124             // 1 board on screen.
0125         scrollChessboard = new JScrollPane(chessboard);
0126         return scrollChessboard;
0127     }
0128     /**
0129     * Player input field for them to select a move.
0130     * @return TextField for player to input moves.
0131     */
0132     public JTextField playerInput()
0133     {
0134         input = new JTextField(10);// More than enough room to write a piece
0135             // and where to move.
0136         input.addActionListener(this);
0137         return input;
0138     }
0139     /**
0140     * Sets the JFrame for the program and places the elements in it.
0141     */
0142     public static void createAndShowGUI()
0143     {
0144         Chessboard fram = new Chessboard();
0145         JFrame frame = new JFrame("Com Sci IB Chess Sim V 0.2");
0146         JPanel contentPane = new JPanel(new GridBagLayout());
0147         JPanel chessboardPanel = new JPanel(new GridBagLayout());
0148         JPanel outputPanel = new JPanel(new GridBagLayout());
0149         JPanel playerMove = new JPanel(new GridBagLayout());
0150         JLabel playerInput = new JLabel("Enter move here: e.g. P4 4,4 ");
0151         JLabel space = new JLabel(" "); // Space
0152             //between button andJTextArea.
0153         GridBagConstraints c = new GridBagConstraints();
0154         playerMove.add(playerInput);
0155         c.gridx=1;
0156         playerMove.add(fram.playerInput(),c);
0157         c.gridx=2;
0158         playerMove.add(space,c );
0159         c.gridx=3;
0160         playerMove.add(fram.moveButton(),c);
0161         frame.setJMenuBar(fram.createJMenuBar());
0162         frame.setContentPane(contentPane);
0163         contentPane.setBackground(Color.WHITE);
0164         chessboardPanel.add(fram.createChessboardText(),c);
0165         chessboardPanel.setBorder(BorderFactory.createCompoundBorder
0166             (BorderFactory.createTitledBorder("Chessboard"),
0167             BorderFactory.createEmptyBorder(5,5,5,5)));
0168             // Gives a fair amount of space between the boarder
0169             // and the TextArea.

```

```

0170     c.gridx=0;
0171     contentPane.add(chessboardPanel,c);
0172     c.gridx=1;
0173     outputPanel.add(fram.createMovesnStats(),c);
0174     outputPanel.setBorder(BorderFactory.createCompoundBorder
0175         (BorderFactory.createTitledBorder("Moves and Player Statistics"),
0176         BorderFactory.createEmptyBorder(5,5,5,5)));
0177         // Gives a fair amount of space between the boarder
0178         // and the TextArea.
0179     contentPane.add(outputPanel,c);
0180     c.gridy=1;
0181     c.gridx=0;
0182     c.gridwidth=3;
0183     playerMove.setBorder(BorderFactory.createCompoundBorder
0184         (BorderFactory.createTitledBorder("Player input"),
0185         BorderFactory.createEmptyBorder(5,120,5,130)));
0186         // Fits the same length as the above two combined.
0187     contentPane.add(playerMove,c);
0188
0189     frame.setVisible(true);
0190     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
0191     frame.pack();
0192 }
0193
0194 public static void main(String[] args) throws IOException
0195 {
0196     try{
0197         UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
0198     }
0199     catch(UnsupportedLookAndFeelException e){}
0200     catch(ClassNotFoundException e){}
0201     catch(InstantiationException e){}
0202     catch(IllegalAccessException e){}
0203         SwingUtilities.invokeLater(new Runnable() {
0204             public void run() {
0205                 createAndShowGUI();});
0206 }
0207 /**
0208  * The code for events such as button presses and
0209  * activation of JMenuItem's.
0210  */
0211 public void actionPerformed(ActionEvent e)// The event being preformed
0212         // IE pushing a button.
0213 {
0214     Chessboard action = new Chessboard();
0215     if(e.getSource()== Lprofile)
0216     {
0217         try{
0218             if(player1==null)
0219             {
0220                 Object playerinput[] = new Object[13];// 13 is the top number of
0221                 // statistic's a player
0222                 // can have.
0223                 playerinput = action.Loadprofile(null);
0224                 if(playerinput[0]!=null)
0225                 {
0226                     player1 = String.valueOf(playerinput[0]);
0227                     for(int i = 1; i<13;i++)// 13 is the top number of
0228                     // statistic's a player can have.
0229                     {
0230                         player1stats[i-1]=Integer.parseInt(
0231                             String.valueOf(playerinput[i]));
0232                     }

```

```

0233     }
0234 }
0235 else if(player2==null)
0236 {
0237     Object playerinput[] = new Object[13]; // 13 is the top number of
0238                                     // statistic's a player
0239                                     // can have.
0240     playerinput = action.Loadprofile(null);
0241     if(playerinput[0]!=null)
0242     {
0243         player2 = String.valueOf(playerinput[0]);
0244         for(int i = 1; i<13;i++) // 13 is the top number of
0245                                 // statistic's a player can have.
0246         {
0247             player2stats[i-1]=Integer.parseInt(
0248                 String.valueOf(playerinput[i]));
0249         }
0250     }
0251 }
0252 }
0253 catch(FileNotFoundException f){}
0254 catch(IOException f){}
0255 }
0256 if(e.getSource()== stats)
0257 {
0258     if(currentPlayer==1) // currentPlayer is equal to 1 when its player
0259                         // 1's turn.
0260     {
0261         if(player1!=null) // All of the below are based on previous
0262                         // knowledge of where each value will be
0263         {
0264             movesnStats.append("Name: " + player1 + "\n");
0265             movesnStats.append(player1 + "'s Wins: " + String.valueOf(
0266                 player1stats[0]) + "\n");
0267             movesnStats.append(player1 + "'s Loses: " + String.valueOf(
0268                 player1stats[1]) + "\n");
0269             movesnStats.append("Number of Pawn's taken by " + player1 +
0270                 ": " + String.valueOf(player1stats[2]) + "\n");
0271             movesnStats.append("Number of Pawn's lost by " + player1 +
0272                 ": " + String.valueOf(player1stats[3]) + "\n");
0273             movesnStats.append("Number of Queen's taken by " + player1 +
0274                 ": " + String.valueOf(player1stats[4]) + "\n");
0275             movesnStats.append("Number of Queen's lost by " + player1 +
0276                 ": " + String.valueOf(player1stats[5]) + "\n");
0277             movesnStats.append("Number of Rook's taken by " + player1 +
0278                 ": " + String.valueOf(player1stats[6]) + "\n");
0279             movesnStats.append("Number of Rook's lost by " + player1 +
0280                 ": " + String.valueOf(player1stats[7]) + "\n");
0281             movesnStats.append("Number of Bishop's taken by " + player1
0282                 + ": " + String.valueOf(player1stats[8]) + "\n");
0283             movesnStats.append("Number of Bishop's lost by " + player1 +
0284                 ": " + String.valueOf(player1stats[9]) + "\n");
0285             movesnStats.append("Number of Knight's taken by " + player1
0286                 + ": " + String.valueOf(player1stats[10]) + "\n");
0287             movesnStats.append("Number of Knight's lost by " + player1 +
0288                 ": " + String.valueOf(player1stats[11]) + "\n");
0289         }
0290     }
0291     if(currentPlayer==0) // currentPlayer is equal to 0 when its player
0292                         // 2's turn.
0293     {
0294         if(player2!=null) // All of the below are based on previous
0295                         // knowledge of where each value will be

```

```

0296     {
0297         movesnStats.append("Name: " + player2 + "\n");
0298         movesnStats.append(player2 + "'s Wins: " + String.valueOf(
0299             player2stats[0]) + "\n");
0300         movesnStats.append(player2 + "'s Loses: " + String.valueOf(
0301             player2stats[1]) + "\n");
0302         movesnStats.append("Number of Pawn's taken by " + player2 +
0303             ": " + player2stats[2] + "\n");
0304         movesnStats.append("Number of Pawn's lost by " + player2 +
0305             ": " + player2stats[3] + "\n");
0306         movesnStats.append("Number of Queen's taken by " + player2 +
0307             ": " + player2stats[4] + "\n");
0308         movesnStats.append("Number of Queen's lost by " + player2 +
0309             ": " + player2stats[5] + "\n");
0310         movesnStats.append("Number of Rook's taken by " + player2 +
0311             ": " + player2stats[6] + "\n");
0312         movesnStats.append("Number of Rook's lost by " + player2 +
0313             ": " + player2stats[7] + "\n");
0314         movesnStats.append("Number of Bishop's taken by " + player2
0315             + ": " + player2stats[8] + "\n");
0316         movesnStats.append("Number of Bishop's lost by " + player2 +
0317             ": " + player2stats[9] + "\n");
0318         movesnStats.append("Number of Knight's taken by " + player2
0319             + ": " + player2stats[10] + "\n");
0320         movesnStats.append("Number of Knight's lost by " + player2 +
0321             ": " + player2stats[11] + "\n");
0322     }
0323 }
0324 }
0325 if(e.getSource()== Nprofile)
0326 {
0327     try{
0328         if(player1==null)
0329         {
0330             Object playerinput[] = new Object[13]; //player name + statistics
0331             playerinput = action.Newprofile();
0332             if(playerinput[0]!=null)
0333             {
0334                 player1 = String.valueOf(playerinput[0]);
0335                 for(int i = 1; i<13;i++)
0336                 {
0337                     player1stats[i-1]=Integer.parseInt(
0338                         String.valueOf(playerinput[i]));
0339                 }
0340             }
0341         }
0342         else if(player2==null)
0343         {
0344             Object playerinput[] = new Object[13]; //player name + statistics
0345             playerinput = action.Newprofile();
0346             if(playerinput[0]!=null)
0347             {
0348                 player2 = String.valueOf(playerinput[0]);
0349                 for(int i = 1; i<13;i++)
0350                 {
0351                     player2stats[i-1]=Integer.parseInt(
0352                         String.valueOf(playerinput[i]));
0353                 }
0354             }
0355         }
0356     }
0357     catch(FileNotFoundException f){}
0358     catch(IOException f){}

```

```

0359     }
0360
0361
0362     if(e.getSource()== Ngame)
0363     {
0364         currentGame = action.Newgame();
0365         board.updateBoard(currentGame);
0366         action.visualBoard(currentGame, chessboard);
0367     }
0368     if(e.getSource()== Sgame)
0369     {
0370         try{
0371             action.Savegame(currentPlayer, board.outputBoard());
0372         }
0373         catch(IOException f){}
0374     }
0375     if(e.getSource()== Lgame)
0376     {
0377         try{
0378             String tempBoard[][]=action.Loadgame();
0379             currentPlayer = Integer.parseInt(tempBoard[8][7]);
0380             tempBoard[8][7]=null;// Removes the currentPlayer from the array
0381             // so it can make the board.
0382             board.updateBoard(tempBoard);
0383             action.visualBoard(board.outputBoard(), chessboard);
0384         }
0385         catch(IOException f){}
0386     }
0387     if(e.getSource() == Lmove)
0388     {
0389         if(pastedMoves.isEmpty()==false)
0390         {
0391             Object lastMove = pastedMoves.pop();
0392             StringTokenizer popst = new StringTokenizer(
0393                 String.valueOf(lastMove));
0394             int xo = Integer.parseInt(popst.nextToken());
0395             int yo = Integer.parseInt(popst.nextToken());
0396             String piece = popst.nextToken() + " " + popst.nextToken();
0397             String takenPiece = popst.nextToken() + " " + popst.nextToken();
0398             for(int i=0;i<8;i++)
0399             {
0400                 for(int j=0;j<8;j++)
0401                 {
0402                     if(currentGame[i][j]!=null)
0403                     {
0404                         if(currentGame[i][j].equals(piece))
0405                         {
0406                             movesAhead.push(i+" "+j+" "+currentGame[i][j]);
0407                             if(!takenPiece.equals("P N"))
0408                             {
0409                                 currentGame[i][j]=takenPiece;
0410                             }
0411                             else
0412                             {
0413                                 currentGame[i][j]=null;
0414                             }
0415                         }
0416                     }
0417                 }
0418             }
0419             if(currentPlayer==1)
0420             {
0421                 currentPlayer=0;

```

```

0422     }
0423     else if(currentPlayer==0)
0424     {
0425         currentPlayer=1;
0426     }
0427     currentGame[xo][yo] = piece;
0428     board.updateBoard(currentGame);
0429     action.visualBoard(board.outputBoard(), chessboard);
0430 }
0431 else
0432 {
0433     JOptionPane.showMessageDialog(frame,
0434         "No more moves before this.");
0435 }
0436 }
0437 if(e.getSource()== Nmove)
0438 {
0439     if(movesAhead.isEmpty()==false)
0440     {
0441         Object lastMove = movesAhead.pop();
0442         StringTokenizer popst = new StringTokenizer(
0443             String.valueOf(lastMove));
0444         int xo = Integer.parseInt(popst.nextToken());
0445         int yo = Integer.parseInt(popst.nextToken());
0446         String piece = popst.nextToken() + " " + popst.nextToken();
0447         for(int i=0;i<8;i++)
0448         {
0449             for(int j=0;j<8;j++)
0450             {
0451                 if(currentGame[i][j]!=null)
0452                 {
0453                     if(currentGame[i][j].equals(piece))
0454                     {
0455                         pastedMoves.push(i+" "+j+" "+currentGame[i][j]);
0456                         currentGame[i][j]=null;
0457                     }
0458                 }
0459             }
0460         }
0461         if(currentPlayer==1)
0462         {
0463             currentPlayer=0;
0464         }
0465         else if(currentPlayer==0)
0466         {
0467             currentPlayer=1;
0468         }
0469         currentGame[xo][yo] = piece;
0470         board.updateBoard(currentGame);
0471         action.visualBoard(board.outputBoard(), chessboard);
0472     }
0473     else
0474     {
0475         JOptionPane.showMessageDialog(frame,
0476             "No more moves ahead of this.");
0477     }
0478 }
0479 if(e.getSource()== delete)
0480 {
0481     if(currentPlayer==1)
0482     {
0483         try{
0484             in = new RandomAccessFile(player1+".pro", "rw");

```

```

0485         player1=null;
0486         in.seek(0);
0487         in.writeBytes("0");
0488     }
0489     catch(IOException f){}
0490 }
0491 else if(currentPlayer==0)
0492 {
0493     try{
0494         in2 = new RandomAccessFile(player2+".pro","rw");
0495         in2.seek(0);
0496         in2.writeBytes("0");
0497         player2=null;
0498     }
0499     catch(IOException f){}
0500 }
0501 }
0502 }
0503 if(e.getSource()==move)
0504 {
0505     StringTokenizer inputText = new StringTokenizer(
0506         input.getText().replaceAll(","," "),");
0507     currentGame = board.outputBoard();
0508     String testPiece = inputText.nextToken();
0509     int pieceMoved=0;
0510     String piece = null;
0511     if(currentPlayer==1)
0512     {
0513         piece = "P1 ";
0514     }
0515     if(currentPlayer==0)
0516     {
0517         piece = "P2 ";
0518     }
0519     if(testPiece.equals("R1") || testPiece.equals("R2")
0520         || testPiece.equals("K1") || testPiece.equals("K2") ||
0521         testPiece.equals("B1") || testPiece.equals("B2") ||
0522         testPiece.equals("Ki") || testPiece.equals("Qu") ||
0523         testPiece.equals("P1") || testPiece.equals("P2") ||
0524         testPiece.equals("P3") || testPiece.equals("P4") ||
0525         testPiece.equals("P5") || testPiece.equals("P6") ||
0526         testPiece.equals("P7") || testPiece.equals("P8"))
0527     {
0528         piece = piece + testPiece;
0529         int xo=0,yo=0;
0530         for(int i=0;i<8;i++)
0531         {
0532             for(int j=0;j<8;j++)
0533             {
0534                 if(currentGame[i][j]!=null)
0535                 {
0536                     if(currentGame[i][j].equals(piece))
0537                     {
0538                         xo=i;
0539                         yo=j;
0540                     }
0541                 }
0542             }
0543         }
0544         int xn = Integer.parseInt(inputText.nextToken());
0545         xn--; // Since the grid is 1-8 and the array is 0-7 i need to
0546             // subtract 1 to reach it.
0547         int yn = Integer.parseInt(inputText.nextToken());

```

```

0548     yn--; // Same as above.
0549
0550     if(currentPlayer==1)
0551     {
0552         if(board.check(board, "P1", testPiece, xn, yn)==false)
0553         {
0554             StringTokenizer outputText = new StringTokenizer(
0555                 board.movePeice(board, xo, yo, xn, yn, currentPlayer));
0556             String outputPlayer = outputText.nextToken();
0557             String outputTaken = outputText.nextToken();
0558             String outputPlayerMoving = outputText.nextToken();
0559             String outputPieceMoved = outputText.nextToken();
0560             // All of the pre-know numbers are the same as the ones in the
0561             // statistics ActionListener.
0562             if(outputPlayer.equals("P1"))
0563             {
0564                 if(outputTaken.equals("P1") || outputTaken.equals("P2") ||
0565                     outputTaken.equals("P3") || outputTaken.equals("P4") ||
0566                     outputTaken.equals("P5") || outputTaken.equals("P6") ||
0567                     outputTaken.equals("P7") || outputTaken.equals("P8"))
0568                 {
0569                     player2stats[2]++;
0570                     player1stats[3]++;
0571                 }
0572                 else if(outputTaken.equals("Qu"))
0573                 {
0574                     player2stats[4]++;
0575                     player1stats[5]++;
0576                 }
0577                 else if(outputTaken.equals("R1") || outputTaken.equals("R2"))
0578                 {
0579                     player2stats[6]++;
0580                     player1stats[7]++;
0581                 }
0582                 else if(outputTaken.equals("B1") || outputTaken.equals("B2"))
0583                 {
0584                     player2stats[8]++;
0585                     player1stats[9]++;
0586                 }
0587                 else if(outputTaken.equals("K1") || outputTaken.equals("K2"))
0588                 {
0589                     player2stats[10]++;
0590                     player1stats[11]++;
0591                 }
0592             }
0593             if(outputPlayer.equals("P2"))
0594             {
0595                 if(outputTaken.equals("P1") || outputTaken.equals("P2") ||
0596                     outputTaken.equals("P3") || outputTaken.equals("P4") ||
0597                     outputTaken.equals("P5") || outputTaken.equals("P6") ||
0598                     outputTaken.equals("P7") || outputTaken.equals("P8"))
0599                 {
0600                     player1stats[2]++;
0601                     player2stats[3]++;
0602                 }
0603                 if(outputTaken.equals("Qu"))
0604                 {
0605                     player1stats[4]++;
0606                     player2stats[5]++;
0607                 }
0608                 else if(outputTaken.equals("R1") || outputTaken.equals("R2"))
0609                 {
0610                     player1stats[6]++;

```

```

0611         player2stats[7]++;
0612     }
0613     else if(outputTaken.equals("B1")||outputTaken.equals("B2"))
0614     {
0615         player1stats[8]++;
0616         player2stats[9]++;
0617     }
0618     else if(outputTaken.equals("K1")||outputTaken.equals("K2"))
0619     {
0620         player1stats[10]++;
0621         player2stats[11]++;
0622     }
0623 }
0624 String outputChecker = outputText.nextToken();
0625 if(outputChecker.equals("YES"))
0626 {
0627     pastedMoves.push(xo + " " + yo + " " + outputPlayerMoving +
0628         " " + outputPieceMoved + " " + outputPlayer + " " +
0629         outputTaken);
0630     int transferPlayer = Integer.parseInt(
0631         outputText.nextToken());
0632     String outputFinal = outputText.nextToken()+" moved to "+
0633         outputText.nextToken()+ " ,"
0634         + outputText.nextToken();
0635     movesnStats.append(outputFinal);
0636     movesnStats.append("\n");
0637     currentPlayer = transferPlayer;
0638     action.visualBoard(board.outputBoard(), chessboard);
0639     try{
0640         action.Saveprofile();
0641     }
0642     catch(IOException f){}
0643     boolean clearAhead = false;// To clear out the movesAhead
0644         // stack if a new move is made.
0645     while(clearAhead==false)// Will keep repeating till
0646         // movesAhead is empty.
0647     {
0648         if(movesAhead.peek()!=null)
0649         {
0650             movesAhead.pop();// Clears the value from the
0651                 // movesAhead stack.
0652         }
0653         else{clearAhead=true;}
0654     }
0655 }
0656 else
0657 {
0658     String outputFinal=outputChecker + " " +
0659         outputText.nextToken()+ " ";
0660     while(outputText.hasMoreTokens())
0661     {
0662         outputFinal = outputFinal + outputText.nextToken()+" ";
0663     }
0664     movesnStats.append(outputFinal + "\n");
0665 }
0666 }
0667 else
0668 {
0669     movesnStats.append("Peice cannot be moved because king is");
0670     movesnStats.append("\nin check\n");
0671 }
0672 }
0673 else if(currentPlayer==0)

```

```

0674     {
0675         if(board.check(board, "P2", testPiece, xn, yn)==false)
0676         {
0677             StringTokenizer outputText = new StringTokenizer(
0678                 board.movePeice(board, xo, yo, xn, yn, currentPlayer));
0679             String outputPlayer = outputText.nextToken();
0680             String outputTaken = outputText.nextToken();
0681             String outputPlayerMoving = outputText.nextToken();
0682             String outputPieceMoved = outputText.nextToken();
0683             // All of the pre-know numbers are the same as the ones in the
0684             // statistics ActionListener.
0685             if(outputPlayer.equals("P1"))
0686             {
0687                 if(outputTaken.equals("P1") || outputTaken.equals("P2") ||
0688                     outputTaken.equals("P3") || outputTaken.equals("P4") ||
0689                     outputTaken.equals("P5") || outputTaken.equals("P6") ||
0690                     outputTaken.equals("P7") || outputTaken.equals("P8"))
0691                 {
0692                     player2stats[2]++;
0693                     player1stats[3]++;
0694                 }
0695                 else if(outputTaken.equals("Qu"))
0696                 {
0697                     player2stats[4]++;
0698                     player1stats[5]++;
0699                 }
0700                 else if(outputTaken.equals("R1") || outputTaken.equals("R2"))
0701                 {
0702                     player2stats[6]++;
0703                     player1stats[7]++;
0704                 }
0705                 else if(outputTaken.equals("B1") || outputTaken.equals("B2"))
0706                 {
0707                     player2stats[8]++;
0708                     player1stats[9]++;
0709                 }
0710                 else if(outputTaken.equals("K1") || outputTaken.equals("K2"))
0711                 {
0712                     player2stats[10]++;
0713                     player1stats[11]++;
0714                 }
0715             }
0716             if(outputPlayer.equals("P2"))
0717             {
0718                 if(outputTaken.equals("P1") || outputTaken.equals("P2") ||
0719                     outputTaken.equals("P3") || outputTaken.equals("P4") ||
0720                     outputTaken.equals("P5") || outputTaken.equals("P6") ||
0721                     outputTaken.equals("P7") || outputTaken.equals("P8"))
0722                 {
0723                     player1stats[2]++;
0724                     player2stats[3]++;
0725                 }
0726                 if(outputTaken.equals("Qu"))
0727                 {
0728                     player1stats[4]++;
0729                     player2stats[5]++;
0730                 }
0731                 else if(outputTaken.equals("R1") || outputTaken.equals("R2"))
0732                 {
0733                     player1stats[6]++;
0734                     player2stats[7]++;
0735                 }
0736                 else if(outputTaken.equals("B1") || outputTaken.equals("B2"))

```

```

0737     {
0738         player1stats[8]++;
0739         player2stats[9]++;
0740     }
0741     else if(outputTaken.equals("K1")||outputTaken.equals("K2"))
0742     {
0743         player1stats[10]++;
0744         player2stats[11]++;
0745     }
0746 }
0747 String outputChecker = outputText.nextToken();
0748 if(outputChecker.equals("YES"))
0749 {
0750     pastedMoves.push(xo + " " + yo + " " + outputPlayerMoving +
0751         " " + outputPieceMoved + " " + outputPlayer + " " +
0752         outputTaken);
0753     int transferPlayer = Integer.parseInt(
0754         outputText.nextToken());
0755     String outputFinal = outputText.nextToken()+" moved to " +
0756         outputText.nextToken()+ ", "
0757         + outputText.nextToken();
0758     movesnStats.append(outputFinal);
0759     movesnStats.append("\n");
0760     currentPlayer = transferPlayer;
0761     action.visualBoard(board.outputBoard(), chessboard);
0762     try{
0763         action.Saveprofile();
0764     }
0765     catch(IOException f){}
0766     boolean clearAhead = false; // To clear out the movesAhead
0767         // stack if a new move is made.
0768     while(clearAhead==false) // Will keep repeating till
0769         // movesAhead is empty.
0770     {
0771         if(movesAhead.peek()!=null)
0772         {
0773             movesAhead.pop();// Clears the value from the
0774                 // movesAhead stack.
0775         }
0776         else{clearAhead=true;}
0777     }
0778 }
0779 else
0780 {
0781     String outputFinal=outputChecker + " " +
0782         outputText.nextToken()+ " ";
0783     while(outputText.hasMoreTokens())
0784     {
0785         outputFinal = outputFinal + outputText.nextToken()+" ";
0786     }
0787     movesnStats.append(outputFinal + "\n");
0788 }
0789 }
0790 else
0791 {
0792     movesnStats.append("Peice cannot be moved because king is");
0793     movesnStats.append("\nin check\n");
0794 }
0795 }
0796 pieceMoved = 1;
0797 if(pieceMoved==1)
0798 {
0799     boolean endGame=false;

```

```

0800     int xKing =0;
0801     int yKing =0;
0802     currentGame = board.outputBoard();
0803     if(piece.equals("P1"))
0804     {
0805         for(int i =0;i<8;i++)
0806         {
0807             for(int j=0;j<8;j++)
0808             {
0809                 if(currentGame[i][j].equals("P2 Ki"))
0810                 {
0811                     xKing = i;
0812                     yKing = j;
0813                 }
0814             }
0815         }
0816     }
0817     else if(piece.equals("P2"))
0818     {
0819         for(int i =0;i<8;i++)
0820         {
0821             for(int j=0;j<8;j++)
0822             {
0823                 if(currentGame[i][j].equals("P1 Ki"))
0824                 {
0825                     xKing = i;
0826                     yKing = j;
0827                 }
0828             }
0829         }
0830     }
0831     endGame = action.checkmate(board, currentGame, piece, xKing,
0832                               yKing);
0833     if(endGame==true&&piece.equals("P1"))
0834     {
0835         try{
0836             movesnStats.append("Gameover, " + player1 + "wins!");
0837             action.Saveprofile();
0838         }
0839         catch(IOException f){}
0840     }
0841     else if(endGame==true&&piece.equals("P2"))
0842     {
0843         try{
0844             movesnStats.append("Gameover, " + player2 + "wins!");
0845             action.Saveprofile();
0846         }
0847         catch(IOException f){}
0848     }
0849 }
0850 }
0851 if(pieceMoved==0)
0852 {
0853     JOptionPane.showMessageDialog(frame, "      Piece does not exist" +
0854     " \n " +"(Note: piece names are CASE SENSITIVE)");
0855 }
0856 }
0857 }
0858 }
0859 /**
0860  * Outputs the board to the Chessboard for so the player can see it.
0861  * @param board the board that JT you want to output
0862  * @param movesnStats the JTextArea being outputed to.

```

```

0863  */
0864  public void visualBoard(String[][] board,      // Inputs the board to be
0865                          // outputed.
0866                          JTextArea movesnStats)// Inputs the TextArea to be
0867                          // outputed to.
0868  {
0869      String rowToOutput=null;
0870      movesnStats.append("  1  2  3  4  5  6  7  8\n");// Number
0871                          // row.
0872      movesnStats.append(" +---+---+---+---+---+---+---+---+\n");// Top row
0873                          // of board.
0874      for(int i=0;i<8;i++)
0875      {
0876          rowToOutput=(i+1) + "|";
0877          for(int j=0;j<8;j++)
0878          {
0879              if(board[i][j]!=null)
0880              {
0881                  StringTokenizer st = new StringTokenizer(board[i][j]);
0882                  String pieceToOutput = null;
0883                  String player = st.nextToken();
0884                  String piece = st.nextToken();
0885                  if(player.equals("P1"))
0886                  {
0887                      if(piece.equals("Queen"))
0888                      {
0889                          pieceToOutput = "WQu";
0890                      }
0891                      else if(piece.equals("King"))
0892                      {
0893                          pieceToOutput = "WKi";
0894                      }
0895                      else
0896                      {
0897                          pieceToOutput = "W"+piece;
0898                      }
0899                  }
0900                  else if(player.equals("P2"))
0901                  {
0902                      if(piece.equals("Qu"))
0903                      {
0904                          pieceToOutput = "BQu";
0905                      }
0906                      else if(piece.equals("Ki"))
0907                      {
0908                          pieceToOutput = "BKi";
0909                      }
0910                      else
0911                      {
0912                          pieceToOutput = "B"+piece;
0913                      }
0914                  }
0915                  rowToOutput = rowToOutput + pieceToOutput + "|";
0916              }
0917              else
0918              {
0919                  rowToOutput = rowToOutput + "  " + "|";
0920              }
0921          }
0922          rowToOutput=rowToOutput+"\n";
0923          movesnStats.append(rowToOutput);
0924          movesnStats.append(" +---+---+---+---+---+---+---+---+\n");//Between
0925                          // rows.

```

```

0926     }
0927     movesnStats.append("\n");
0928 }
0929 /**
0930  * Outputs the game to a file that can be loaded later.
0931  * @param savegame the game u want to save
0932  * @throws IOException
0933  */
0934 public void Savegame(int currentPlayer, // Saves whose turn it was when
0935                     // the game was saved.
0936                     String savegame[][])// The current board when the game
0937                     // was saved.
0938     throws IOException
0939 {
0940 {
0941     String game[][] = savegame;
0942     StringTokenizer inst = null;
0943     save.setDialogType(JFileChooser.SAVE_DIALOG);
0944     int returnVal = save.showOpenDialog(frame);
0945     if(returnVal == JFileChooser.APPROVE_OPTION)
0946     {
0947         inst = new StringTokenizer(
0948             save.getSelectedFile().getName().replace(".", " "));
0949         inst.nextToken();// Removes the files, name extention is all that is
0950             // needed right now.
0951         String fileExtention = inst.nextToken();
0952         if(!fileExtention.equals("gam"))
0953         {
0954             out = new PrintWriter(
0955                 new BufferedWriter(
0956                     new FileWriter(
0957                         save.getName(save.getSelectedFile()+".gam")));
0958             }
0959         else if(fileExtention.equals("gam"))
0960         {
0961             out = new PrintWriter(
0962                 new BufferedWriter(
0963                     new FileWriter(
0964                         save.getName(save.getSelectedFile()))));
0965             }
0966         out.print(currentPlayer + " ");
0967         for(int x=0;x<8;x++)
0968         {
0969             for(int y=0;y<8;y++)
0970             {
0971                 if(game[x][y]!=null)
0972                 {
0973                     out.print(x+" "+y+" "+game[x][y]+" ");
0974                 }
0975             }
0976         }
0977     }
0978     out.close();
0979 }
0980 /**
0981  * Sets up a new game.
0982  * @return newGame An array containing the pieces starting locations.
0983  */
0984 public String[][] Newgame()
0985 {
0986     // All of the below numbers are the known locations of the pieces at
0987     the
0988     // start of a game.

```

```

0988     String newGame[][] = new String[8][8];
0989     newGame[0][0] = "P1 R1";
0990     newGame[0][1] = "P1 K1";
0991     newGame[0][2] = "P1 B1";
0992     newGame[0][3] = "P1 Qu";
0993     newGame[0][4] = "P1 Ki";
0994     newGame[0][5] = "P1 B2";
0995     newGame[0][6] = "P1 K2";
0996     newGame[0][7] = "P1 R2";
0997     newGame[1][0] = "P1 P1";
0998     newGame[1][1] = "P1 P2";
0999     newGame[1][2] = "P1 P3";
1000     newGame[1][3] = "P1 P4";
1001     newGame[1][4] = "P1 P5";
1002     newGame[1][5] = "P1 P6";
1003     newGame[1][6] = "P1 P7";
1004     newGame[1][7] = "P1 P8";
1005
1006     newGame[7][0] = "P2 R1";
1007     newGame[7][1] = "P2 K1";
1008     newGame[7][2] = "P2 B1";
1009     newGame[7][3] = "P2 Qu";
1010     newGame[7][4] = "P2 Ki";
1011     newGame[7][5] = "P2 B2";
1012     newGame[7][6] = "P2 K2";
1013     newGame[7][7] = "P2 R2";
1014     newGame[6][0] = "P2 P1";
1015     newGame[6][1] = "P2 P2";
1016     newGame[6][2] = "P2 P3";
1017     newGame[6][3] = "P2 P4";
1018     newGame[6][4] = "P2 P5";
1019     newGame[6][5] = "P2 P6";
1020     newGame[6][6] = "P2 P7";
1021     newGame[6][7] = "P2 P8";
1022     currentPlayer = 1;
1023     return newGame;
1024 }
1025 /**
1026  * allows players to load previously saved games.
1027  * @return the previously saved game.
1028  * @throws FileNotFoundException
1029  */
1030 public String[][] Loadgame() throws FileNotFoundException, IOException
1031 {
1032     String loadboard[][] = new String[9][8]; // the 9 Allows the output of
1033                                             // which players turn it was prior
1034                                             // to saving the 8 is the same as it was
1035                                             // at the top of the program.
1036     try{
1037         int returnVal = load.showOpenDialog(frame);
1038         if(returnVal == JFileChooser.APPROVE_OPTION)
1039         {
1040             in3 = new RandomAccessFile(load.getSelectedFile().getName(), "r");
1041         }
1042         StringTokenizer st = new StringTokenizer(in3.readLine());
1043         loadboard[8][7] = st.nextToken(); // The player whose turn it currently
1044                                         // will take this spot.
1045         while(st.hasMoreTokens())
1046         {
1047             int row = Integer.parseInt(st.nextToken());
1048             int column = Integer.parseInt(st.nextToken());
1049             String peice = st.nextToken() + " " + st.nextToken();
1050             loadboard[row][column] = peice;

```

```

1051     }
1052     return loadboard;
1053 }
1054 }
1055 catch(FileNotFoundException e){ JOptionPane.showMessageDialog(
1056     frame, "FileNotFoundException: " + e.getMessage());}
1057 catch(IOException e){ JOptionPane.showMessageDialog(
1058     frame, "IOException: " + e.getMessage());}
1059 return loadboard;
1060 }
1061 /**
1062  * Allows the user to load a premade profile with all of their
1063  * individual statistic's.
1064  * @return object array containing player 1's or player 2's information.
1065  * @throws FileNotFoundException
1066  * @throws IOException
1067  */
1068 public Object[] Loadprofile(String newprofile // If a new profile is
1069     // being loaded.
1070     ) throws FileNotFoundException, IOException
1071 {
1072     Object firstplayer[] = new Object[13]; // Name+all statistics
1073     // totaling 13
1074     Object secondplayer[] = new Object[13]; // Name+all statistics
1075     // totaling 13
1076     Object outputarray[] = new Object[13]; // Will contain 1 of the two
1077     // above to simplify output
1078     newprofilename = newprofile;
1079     if(newprofilename!=null)
1080     {
1081         if(player1==null)
1082         {
1083             player1=newprofilename;
1084             try{
1085                 in = new RandomAccessFile(player1+".pro", "r");
1086             }
1087             catch(FileNotFoundException e)
1088             { JOptionPane.showMessageDialog(
1089                 frame, "FileNotFoundException: " + e.getMessage());}
1090             StringTokenizer st = new StringTokenizer(
1091                 in.readLine().replaceAll("_", " "));
1092             st.nextToken();//jumps the deleted token
1093             st.nextToken(); //jumps the name token
1094             firstplayer[0]=player1;
1095             for(int i = 1;i<13;i++)
1096             {
1097                 firstplayer[i]=st.nextToken();
1098             }
1099
1100             outputarray = firstplayer;
1101         }
1102     }
1103     if(player2==null)
1104     {
1105         player2=newprofilename;
1106         try{
1107             in2 = new RandomAccessFile(player2+".pro", "r");
1108         }
1109         catch(FileNotFoundException e)
1110         { JOptionPane.showMessageDialog(
1111             frame, "FileNotFoundException: " + e.getMessage());}
1112         StringTokenizer st = new StringTokenizer(
1113             in2.readLine().replaceAll("_", " "));

```

```

1114         st.nextToken();//jumps the deleted token
1115         st.nextToken(); //jumps the name token
1116         secondplayer[0]=player2;
1117         for(int i = 1;i<13;i++)
1118         {
1119             secondplayer[i]=st.nextToken();
1120         }
1121         outputarray = secondplayer;
1122     }
1123 }
1124 }
1125 }
1126 if(newprofilename==null)
1127 {
1128     if(player1==null)
1129     {
1130         int returnVal = load.showOpenDialog(frame);
1131         if(returnVal == JFileChooser.APPROVE_OPTION) {
1132             try{
1133                 StringTokenizer inst =
1134                     new StringTokenizer(
1135                         load.getSelectedFile()
1136                             .getName().replace(".", " "));
1137                 inst.nextToken();//Removes the files, name extension
1138                 // is all that is needed right now.
1139                 String fileExtention = inst.nextToken();
1140                 if(fileExtention.equals("pro"))
1141                 {
1142                     in = new RandomAccessFile(
1143                         load.getSelectedFile().getName(),"r");
1144                     StringTokenizer st =
1145                         new StringTokenizer(
1146                             in.readLine()
1147                                 .replaceAll("_", " ").trim());
1148                     if(st.nextToken().equals("0"))
1149                     {
1150                         JOptionPane.showMessageDialog(
1151                             frame, "Profile has been deleted.");
1152                         in.close();
1153                     }
1154                     else
1155                     {
1156                         firstplayer[0] = st.nextToken();
1157                         for(int i=1;i<13;i++)
1158                         {
1159                             firstplayer[i]=st.nextToken();
1160                         }
1161                     }
1162                 }
1163                 if(!fileExtention.equals("pro"))
1164                 {JOptionPane.showMessageDialog(
1165                     frame, "File Selected is not a profile.");}
1166             }
1167             catch(FileNotFoundException e)
1168             { JOptionPane.showMessageDialog(frame,
1169                 "FileNotFoundException: " + e.getMessage());}
1170             catch(IOException e)
1171             { JOptionPane.showMessageDialog(
1172                 frame, "IOException: " + e.getMessage());}
1173         }
1174         outputarray = firstplayer;
1175     }
1176     else

```

```

1177     {
1178         int returnVal = load.showOpenDialog(frame);
1179         if(returnVal == JFileChooser.APPROVE_OPTION) {
1180             try{
1181                 if(load.getSelectedFile().getName()!=null)
1182                 {
1183                     StringTokenizer inst =
1184                         new StringTokenizer(
1185                             load.getSelectedFile()
1186                                 .getName().replace(".", " "));
1187                     inst.nextToken();// Removes the files name extension
1188                         // is all that is needed right now.
1189                     String fileExtention = inst.nextToken();
1190                     if(fileExtention.equals("pro"))
1191                     {
1192                         in2 = new RandomAccessFile(
1193                             load.getSelectedFile().getName(),"r");
1194                         StringTokenizer st =
1195                             new StringTokenizer(
1196                                 in2.readLine()
1197                                     .replaceAll("_", " ").trim());
1198                         if(st.nextToken().equals("0"))
1199                         {
1200                             JOptionPane.showMessageDialog(
1201                                 frame, "Profile has been deleted.");
1202                             in2.close();
1203                         }
1204                         else
1205                         {
1206                             secondplayer[0] = st.nextToken();
1207                             for(int i=1;i<13;i++)
1208                             {
1209                                 secondplayer[i]=st.nextToken();
1210                             }
1211                         }
1212                     }
1213                     if(!fileExtention.equals("pro"))
1214                     {JOptionPane.showMessageDialog(
1215                         frame, "File Selected is not a profile.");}
1216                 }
1217             }
1218             catch(FileNotFoundException e)
1219             { JOptionPane.showMessageDialog(
1220                 frame, "FileNotFoundException: " + e.getMessage());}
1221             catch(IOException e)
1222             { JOptionPane.showMessageDialog(
1223                 frame, "FileNotFoundException: " + e.getMessage());}
1224         }
1225         outputarray = secondplayer;
1226     }
1227 }
1228 return outputarray;
1229 }
1230 /**
1231  * Updates the players profile after each game or when a game is saved.
1232  * @throws IOException
1233  */
1234 public void Saveprofile() throws IOException
1235 {
1236     if(player1!=null)
1237     {
1238         in = new RandomAccessFile(player1+".pro","rw");
1239         in.seek(27);//skips the players name.

```

```

1240     for(int i=0;i<13;i++)
1241     {
1242         int maxint = 5;// Used to standardize the length of output
1243         // numbers.
1244         String player1stat = String.valueOf(player1stats[i]);
1245         while(player1stat.length()<=maxint)
1246             {player1stat = player1stat+"_";}
1247         in.writeBytes(player1stat);
1248     }
1249 }
1250 if(player2!=null)
1251 {
1252     in = new RandomAccessFile(player2+".pro", "rw");
1253     in.seek(27);//skips the players name.
1254     for(int i=0;i<13;i++)
1255     {
1256         int maxint = 5;
1257         String player2stat = String.valueOf(player2stats[i]);
1258         while(player2stat.length()<=maxint)
1259             {player2stat = player2stat+"_";}
1260         in.writeBytes(player2stat);
1261     }
1262 }
1263 }
1264 /**
1265  * Sets up a players new profile and loads it into either player1
1266  * or player2 using Loadprofile.
1267  * @throws IOException
1268  * @retrun the new player information
1269  */
1270 public Object[] Newprofile() throws IOException
1271 {
1272     Object playerinput[] = new Object[14];
1273     Chessboard newplayer = new Chessboard();
1274     save.setDialogType(JFileChooser.OPEN_DIALOG);
1275     int returnVal = save.showOpenDialog(frame);
1276     if(returnVal == JFileChooser.APPROVE_OPTION)
1277     {
1278         try{
1279             String inputname = save.getName(save.getSelectedFile());
1280             if(inputname!=null)
1281             {
1282                 in = new RandomAccessFile(inputname+".pro", "rw");
1283                 newprofilename = "1_" + save.getName(save.getSelectedFile())
1284                     .replaceAll(".pro", " ").trim();
1285                 int count = newprofilename.length();
1286                 while(count<27){newprofilename=newprofilename+"_";count++;}
1287                 in.setLength(87);// 27 for name, 6 for colour, and 5 for
1288                     // each score catagory.
1289                 in.writeBytes(newprofilename);
1290                 for(int i=0;i<12;i++)
1291                 {
1292                     in.writeBytes("0_____");
1293                 }
1294
1295                 if(player1==null)
1296                 {
1297                     playerinput = newplayer.Loadprofile(inputname);
1298                 }
1299                 else if(player2==null)
1300                 {
1301                     playerinput = newplayer.Loadprofile(inputname);
1302                 }

```

```

1303     }
1304     }
1305     catch(IOException e){JOptionPane.showMessageDialog(
1306         frame, "IOException: " + e.getMessage());}
1307 }
1308 return playerinput;
1309 }
1310 }
1311 }

/**=====
 * Class:      ChessMoves      IB Dossier      Author: Mathew George
 * Version:    001              Date:   November 9, 2007
 *
 * Purpose     Contains the game mechanics for a chess simulator.
 *
 * Course:     Computer Science 33 IB          Teacher:  Gerry Donaldson
 *
 * School:     Sir Winston Churchill High School, Calgary, Alberta, Canada
 *
 * Language:   Java J2SE 6.0 Target Operating System: Java Virtual Machine
 *
 * System:     Intel Celeron 3GHz running under Windows XP IDE: Eclipse 3.2
 *
=====*/

0001 package src;
0002 import java.util.*;
0003
0004 public class ChessMoves
0005 {
0006     /**
0007      * Constructor of the program that builds a new board that can
0008      * be updated and allows for the player to move pieces.
0009      */
0010     public ChessMoves()
0011     {
0012         board = new String[8][8];
0013     }
0014     /**
0015      * Updates the information of the board by inputting a new board.
0016      * @param inputboard contains the board
0017      */
0018     public void updateBoard(String inputboard[][])// The board replacing the
0019         // old 1.
0020     {
0021         board=inputboard;
0022     }
0023     /**
0024      * Outputs the board containing the pieces locations.
0025      * @return board
0026      */
0027     public String[][] outputBoard()
0028     {
0029         return board;
0030     }
0031     /**
0032      * Checks if the piece being moved exists and if its a legal move.
0033      * @param peice
0034      * @param xo
0035      * @param yo

```

```

0036 * @param xn
0037 * @param yn
0038 * @param currentPlayer
0039 * @return returnMove contains whose turn it now is, where the piece
0040 *         was, where it is now, and if it took a piece.
0041 */
0042 public String movePeice(ChessMoves peice, // So the board and other
0043                        // methods within ChesMoves can
0044                        // be called.
0045                        int xo, //The locations of the piece.
0046                        int yo, //same as above.
0047                        int xn, //The location the piece is moving.
0048                        int yn, //same as above.
0049                        int currentPlayer) //the Player moving.
0050 {
0051     boolean moveable=false;
0052     boolean rightPlayer = false;
0053     String returnMove = null;
0054     String player=null;
0055     String peiceToMove = null;
0056     String playerOnSpace=null;
0057     if(peice.isPeiceThere(xo, yo)==true)
0058     {
0059         StringTokenizer st = new StringTokenizer(board[xo][yo]);
0060         player = st.nextToken();
0061         peiceToMove = st.nextToken();
0062         if(currentPlayer==1&&player.equals("P1"))
0063         {
0064             rightPlayer=true;
0065         }
0066         else if(currentPlayer==0&&player.equals("P2"))
0067         {
0068             rightPlayer=true;
0069         }
0070         if(rightPlayer==true)
0071         {
0072             if(peiceToMove.equals("R1")||peiceToMove.equals("R2"))
0073             {
0074                 if(yn==yo)
0075                 {
0076                     moveable = peice.possibleColumnMoves(peice, player, xn, xo, yo);
0077                 }
0078                 else if(xn==xo)
0079                 {
0080                     moveable = peice.possibleRowMoves(peice, player, yn, yo, xo);
0081                 }
0082             }
0083             if(peiceToMove.equals("K1")||peiceToMove.equals("K2"))
0084             {
0085                 moveable = peice.possibleKnightMoves(
0086                     peice, player, yn, yo, xn, xo);
0087             }
0088             if(peiceToMove.equals("B1")||peiceToMove.equals("B2"))
0089             {
0090                 moveable = peice.possibleDiagonalMoves(
0091                     peice, player, xo, xn, yo, yn);
0092             }
0093             if(peiceToMove.equals("Qu"))
0094             {
0095                 moveable = peice.possibleColumnMoves(peice, player, xn, xo, yo);
0096                 if(moveable!=true){moveable = peice.possibleRowMoves(
0097                     peice, player, yn, yo, xo);
0098                 if(moveable!=true){moveable = peice.possibleDiagonalMoves(

```

```

0099         peice, player, xo, xn, yo, yn);}
0100     }
0101 }
0102 if(peiceToMove.equals("Ki"))
0103 {
0104     moveable = peice.possibleKingMoves(
0105         peice, player, yn, yo, xn, xo);
0106 }
0107 if(peiceToMove.equals("P1") || peiceToMove.equals("P2") ||
0108     peiceToMove.equals("P3") || peiceToMove.equals("P4") ||
0109     peiceToMove.equals("P5") || peiceToMove.equals("P6") ||
0110     peiceToMove.equals("P7") || peiceToMove.equals("P8"))
0111 {
0112     moveable = peice.possiblePawnMoves(
0113         peice, player, peiceToMove, yn, yo, xn, xo);
0114 }
0115 }
0116 if(moveable==true)
0117 {
0118     if(board[xn][yn]!=null)
0119     {
0120         st = new StringTokenizer(board[xn][yn]);
0121         playerOnSpace = st.nextToken();
0122     }
0123
0124     if(peice.isPeiceThere(xn, yn)!=true || !playerOnSpace.equals(player))
0125     {
0126         if(playerOnSpace!=null)
0127         {
0128             if(!playerOnSpace.equals(player))
0129             {
0130                 returnMove = board[xn][yn];
0131             }
0132         }
0133         else{returnMove = "P N";}
0134         board[xn][yn]= board[xo][yo];
0135         board[xo][yo]=null;
0136         returnMove = returnMove + " " + board[xn][yn];
0137     }
0138 }
0139 }
0140 if(moveable==true)
0141 {
0142     returnMove = returnMove + " YES ";
0143     if(player.equals("P1"))
0144     {
0145         returnMove = returnMove + "0 W";
0146     }
0147     else if(player.equals("P2"))
0148     {
0149         returnMove = returnMove + "1 B";
0150     }
0151     if(peiceToMove.equals("King"))
0152     {
0153         returnMove = returnMove + "Ki";
0154     }
0155     else if(peiceToMove.equals("Queen"))
0156     {
0157         returnMove = returnMove + "Qu";
0158     }
0159     else
0160     {
0161         returnMove = returnMove + peiceToMove+ " "+(xn+1)+" "+ (yn+1);

```

```

0162     }
0163     }
0164     if(moveable==false)
0165     {
0166         returnMove = "P N NO Peice cannot be moved there.";
0167     }
0168     }
0169     return returnMove;
0170 }
0171 /**
0172  * Checks if a piece is on the specified position
0173  * @param x
0174  * @param y
0175  * @return true if a piece is there, false if there isn't
0176  */
0177 public boolean isPeiceThere(int x,//Location of Piece.
0178                          int y)//same as above.
0179 {
0180     if(board[x][y]!=null){return true;}
0181     else return false;
0182 }
0183 /**
0184  * Checks if a piece can move within the column its on.
0185  * @param peice
0186  * @param playermoving
0187  * @param xn
0188  * @param xo
0189  * @param yo
0190  * @return true if it can false if it can't
0191  */
0192 public boolean possibleColumnMoves(ChessMoves peice, // So the board
0193                                   // and other methods
0194                                   // within ChesMoves
0195                                   // can be called.
0196                                   String playermoving, // Player that is
0197                                   // moving.
0198                                   int xn,           // row piece is
0199                                   // moving to.
0200                                   int xo,           // location of piece
0201                                   // thats moving.
0202                                   int yo)           // same as above.
0203 {
0204     boolean moveable = false;
0205     boolean inTransition = true;
0206     if(xn>xo)
0207     {
0208         xo++;
0209         while(xn>=xo)
0210         {
0211             if(xn!=xo)
0212             {
0213                 if(peice.isPeiceThere(xo, yo)==true)
0214                 {
0215                     inTransition = false;
0216                 }
0217             }
0218             else if(xn==xo&&inTransition==true)
0219             {
0220                 moveable = true;
0221             }
0222             xo++;
0223         }
0224     }

```

```

0225     else if(xn<=xo)
0226     {
0227         xo--;
0228         while(xn<=xo)
0229         {
0230             if(xn!=xo)
0231             {
0232                 if(peice.isPeiceThere(xo, yo)==true)
0233                 {
0234                     inTransition = false;
0235                 }
0236             }
0237             else if(xn==xo&&inTransition==true)
0238             {
0239                 moveable = true;
0240             }
0241             xo--;
0242         }
0243     }
0244     return moveable;
0245 }
0246 /**
0247  * Checks if the piece can move within the row its on does not include
0248  * castling.
0249  * @param peice
0250  * @param playermoving
0251  * @param yn
0252  * @param yo
0253  * @param xn
0254  * @return true if it can, false if it can't
0255  */
0256 public boolean possibleRowMoves(ChessMoves peice, // So the board and
0257                                 // other methods
0258                                 // within ChesMoves
0259                                 // can be called.
0260                                 String playermoving, // Player that is
0261                                 // moving.
0262                                 int yn, // column piece is
0263                                 // moving to.
0264                                 int yo, // location of the
0265                                 // piece thats
0266                                 // moving.
0267                                 int xn) // same as above.
0268 {
0269     boolean moveable = false;
0270     boolean inTransition = true;
0271     if(yn>yo)
0272     {
0273         yo++;
0274         while(yn>=yo&&inTransition==true)
0275         {
0276             if(yn!=yo)
0277             {
0278                 if(peice.isPeiceThere(xn, yo)==true)
0279                 {
0280                     inTransition = false;
0281                 }
0282             }
0283             else if(yn==yo&&inTransition==true)
0284             {
0285                 moveable = true;
0286             }
0287             yo++;

```

```

0288     }
0289     }
0290     else if(yn<=yo&&inTransition==true)
0291     {
0292         yo--;
0293         while(yn<=yo)
0294         {
0295             if(yn!=yo)
0296             {
0297                 if(peice.isPeiceThere(xn, yo)==true)
0298                 {
0299                     inTransition = false;
0300                 }
0301             }
0302             else if(yn==yo&&inTransition==true)
0303             {
0304                 moveable = true;
0305             }
0306             yo--;
0307         }
0308     }
0309     return moveable;
0310 }
0311 /**
0312  * Checks if a piece can move diagonally (any of the four directions)
0313  * based on its location.
0314  * @param peice
0315  * @param player
0316  * @param xo
0317  * @param xn
0318  * @param yo
0319  * @param yn
0320  * @return true if it can, false if it can't
0321  */
0322 public boolean possibleDiagonalMoves(ChessMoves peice, // So the board
0323                                     // and other methods
0324                                     // within ChesMoves
0325                                     // can be called.
0326                                     String player, // Player that is
0327                                     // moving.
0328                                     int xo, // Location of the
0329                                     // piece moving.
0330                                     int xn, // Location the
0331                                     // piece is moving
0332                                     // to.
0333                                     int yo, // Location of the
0334                                     // piece moving.
0335                                     int yn) // Location the
0336                                     // piece is moving
0337                                     // to.
0338 {
0339     boolean moveable = false;
0340     boolean inTransition = true;
0341     if(xn>xo&&yn>yo)
0342     {
0343         while(xn>xo&&yn>yo&&inTransition==true)
0344         {
0345             xo++;
0346             yo++;
0347             if(peice.isPeiceThere(xo, yo)==true&&xo!=xn&&yn!=yo)
0348             {
0349                 inTransition = false;
0350             }
0351         }
0352     }

```

```

0351         if(xn==xo&&yn==yo&&inTransition==true)
0352         {
0353             moveable = true;
0354         }
0355     }
0356 }
0357 else if(xn>xo&&yn<yo)
0358 {
0359     while(xn>xo&&yn<yo&&inTransition==true)
0360     {
0361         xo++;
0362         yo--;
0363         if(peice.isPeiceThere(xo, yo)==true&&xo!=xn&&yn!=yo)
0364         {
0365             inTransition = false;
0366         }
0367         if(xn==xo&&yn==yo&&inTransition==true)
0368         {
0369             moveable = true;
0370         }
0371     }
0372 }
0373 else if(xn<xo&&yn>yo)
0374 {
0375     while(xn<xo&&yn>yo&&inTransition==true)
0376     {
0377         xo--;
0378         yo++;
0379         if(peice.isPeiceThere(xo, yo)==true&&xo!=xn&&yn!=yo)
0380         {
0381             inTransition = false;
0382         }
0383         if(xn==xo&&yn==yo&&inTransition==true)
0384         {
0385             moveable = true;
0386         }
0387     }
0388 }
0389 else if(xn<xo&&yn<yo)
0390 {
0391     while(xn<xo&&yn<yo&&inTransition==true)
0392     {
0393         xo--;
0394         yo--;
0395         if(peice.isPeiceThere(xo, yo)==true&&xo!=xn&&yn!=yo)
0396         {
0397             inTransition = false;
0398         }
0399         if(xn==xo&&yn==yo&&inTransition==true)
0400         {
0401             moveable = true;
0402         }
0403     }
0404 }
0405 return moveable;
0406 }
0407 /**
0408  * Checks if the move selected is legal for a Pawn to make includes
0409  * En passant but not promotion
0410  * @param peice
0411  * @param playermoving
0412  * @param peiceToMove
0413  * @param yn

```

```

0414 * @param yo
0415 * @param xn
0416 * @param xo
0417 * @return true if it can, false if it can't
0418 */
0419 public boolean possiblePawnMoves(ChessMoves peice, // So the board
0420                                 // and other methods
0421                                 // within ChesMoves
0422                                 // can be called.
0423                                 String playermoving, // Player that
0424                                 // is moving.
0425                                 String peiceToMove, // Specific Pawn
0426                                 // thats moving.
0427                                 int yn, // Location the pawn
0428                                 // is moving to.
0429                                 int yo, // Current location
0430                                 // of the pawn.
0431                                 int xn, // Location the pawn
0432                                 // is moving to.
0433                                 int xo) // Current location
0434                                 // of the pawn.
0435 {
0436     boolean moveable = false;
0437     boolean firstPawnMove = false;
0438     if(playermoving.equals("P1"))
0439     {
0440         for(int i=0;i<8;i++)
0441         {
0442             String peices = "P"+(i+1);
0443             if(peiceToMove.equals(peices)&&xo==1)
0444             {
0445                 firstPawnMove = true;
0446             }
0447         }
0448     }
0449     else if(playermoving.equals("P2")&&xo==6)
0450     {
0451         for(int i=0;i<8;i++)
0452         {
0453             if(peiceToMove.equals("P"+(i+1)))
0454             {
0455                 firstPawnMove = true;
0456             }
0457         }
0458     }
0459     if(xn<xo&&yn==yo&&playermoving.equals("P2"))
0460     {
0461         xo--;
0462         if(xn==xo&&yn==yo)
0463         {
0464             moveable=true;
0465         }
0466         xo++;
0467     }
0468     if(xn<xo&&yn>yo&&playermoving.equals("P2"))
0469     {
0470         xo--;
0471         yo++;
0472         if(xn==xo&&yn==yo&&peice.isPeiceThere(xn, yn)==true)
0473         {
0474             moveable=true;
0475         }
0476         xo++;

```

```

0477     yo--;
0478 }
0479 if(xn<xo&&yn<yo&&playermoving.equals("P2"))
0480 {
0481     xo--;
0482     yo--;
0483     if(xn==xo&&yn==yo&&peice.isPeiceThere(xn, yn)==true)
0484     {
0485         moveable=true;
0486     }
0487     xo++;
0488     yo--;
0489 }
0490 if(xn<xo&&yo==yn&&playermoving.equals("P2")&&firstPawnMove==true)
0491 {
0492     xo=xo-2;
0493     if(xn==xo&&yn==yo)
0494     {
0495         moveable=true;
0496     }
0497     xo=xo+2;
0498 }
0499 if(xn>xo&&yn==yo&&playermoving.equals("P1"))
0500 {
0501     xo++;
0502     if(xn==xo&&yn==yo)
0503     {
0504         moveable=true;
0505     }
0506     xo--;
0507 }
0508 if(xn>xo&&yn>yo&&playermoving.equals("P1"))
0509 {
0510     xo++;
0511     yo++;
0512     if(xn==xo&&yn==yo&&peice.isPeiceThere(xn, yn)==true)
0513     {
0514         moveable=true;
0515     }
0516     xo--;
0517     yo--;
0518 }
0519 else if(xn>xo&&yn<yo&&playermoving.equals("P1"))
0520 {
0521     xo++;
0522     yo--;
0523     if(xn==xo&&yn==yo&&peice.isPeiceThere(xn, yn)==true)
0524     {
0525         moveable=true;
0526     }
0527     xo--;
0528     yo++;
0529 }
0530 if(xn>xo&&yo==yn&&playermoving.equals("P1")&&firstPawnMove==true)
0531 {
0532     xo=xo+2;
0533     if(xn==xo&&yn==yo)
0534     {
0535         moveable=true;
0536     }
0537 }
0538 return moveable;
0539 }

```

```

0540 /**
0541  * Checks all of the possible king moves including if he would
0542  * be in check, does not include castling.
0543  * @param peice
0544  * @param playermoving
0545  * @param yn
0546  * @param yo
0547  * @param xn
0548  * @param xo
0549  * @return true if he can, false if he can't
0550  */
0551 public boolean possibleKingMoves(ChessMoves peice, // So the board
0552                                 // and other methods within
0553                                 // ChesMoves can be
0554                                 // called.
0555                                 String playermoving, // The player thats
0556                                 // moving.
0557                                 int yn, // Location the piece
0558                                 // is moving to.
0559                                 int yo, // Current location of
0560                                 // the piece.
0561                                 int xn, // Location the piece
0562                                 // is moving to.
0563                                 int xo) // Current location of
0564                                 // the piece.
0565 {
0566     boolean moveable = false;
0567     if(yn==yo)
0568     {
0569         if(xn>xo)
0570         {
0571             xo++;
0572             if(xo==xn)
0573             {
0574                 moveable = true;
0575             }
0576         }
0577         else if(xn<xo)
0578         {
0579             xo--;
0580             if(xo==xn)
0581             {
0582                 moveable = true;
0583             }
0584         }
0585     }
0586     else if(xn==xo)
0587     {
0588         if(yn>yo)
0589         {
0590             yo++;
0591             if(yn==yo)
0592             {
0593                 moveable = true;
0594             }
0595         }
0596         if(yn<yo)
0597         {
0598             yo--;
0599             if(yn==yo)
0600             {
0601                 moveable = true;
0602             }

```

```

0603     }
0604     }
0605     else if(xn>xo&&yn>yo)
0606     {
0607         xo++;
0608         yo++;
0609         if(xn==xo&&yn==yo)
0610         {
0611             moveable = true;
0612         }
0613     }
0614     else if(xn>xo&&yn<yo)
0615     {
0616         xo++;
0617         yo--;
0618         if(xn==xo&&yn==yo)
0619         {
0620             moveable = true;
0621         }
0622     }
0623     else if(xn<xo&&yn>yo)
0624     {
0625         xo--;
0626         yo++;
0627         if(xn==xo&&yn==yo)
0628         {
0629             moveable = true;
0630         }
0631     }
0632     else if(xn<xo&&yn<yo)
0633     {
0634         xo--;
0635         yo--;
0636         if(xn==xo&&yn==yo)
0637         {
0638             moveable = true;
0639         }
0640     }
0641     return moveable;
0642 }
0643 /**
0644  * Checks all of the possible moves a knight can make.
0645  * @param peice
0646  * @param playermoving
0647  * @param yn
0648  * @param yo
0649  * @param xn
0650  * @param xo
0651  * @return true if he can move, false if he can't
0652  */
0653 public boolean possibleKnightMoves(ChessMoves peice, // So the board
0654                                     // and other methods
0655                                     // within ChesMoves
0656                                     // can be called.
0657                                     String playermoving, // The player that
0658                                     // is moving.
0659                                     int yn, // The location the
0660                                     // piece is moving
0661                                     // to.
0662                                     int yo, // The current
0663                                     // location of
0664                                     // the piece.
0665                                     int xn, // The location the

```

```

0666                                     // piece is moving
0667                                     // to.
0668                                     int xo) // The current
0669                                     // location of the
0670                                     // piece.
0671 {
0672     boolean moveable = false;
0673     if(xn>xo)
0674     {
0675         xo++;
0676         if(xn==xo)
0677         {
0678             if(yn>yo)
0679             {
0680                 yo=yo+2;
0681                 if(yo==yn)
0682                 {
0683                     if(peice.isPeiceThere(xn, yn)==true)
0684                     {
0685                         StringTokenizer st =
0686                             new StringTokenizer(board[xn][yn]);
0687                         if(!playermoving.equals(st.nextToken()))
0688                         {
0689                             moveable=true;
0690                         }
0691                     }
0692                     else {moveable=true;}
0693                 }
0694             }
0695             else if(yn<yo)
0696             {
0697                 yo=yo-2;
0698                 if(yo==yn)
0699                 {
0700                     if(peice.isPeiceThere(xn, yn)==true)
0701                     {
0702                         StringTokenizer st = new StringTokenizer(
0703                             board[xn][yn]);
0704                         if(!playermoving.equals(st.nextToken()))
0705                         {
0706                             moveable=true;
0707                         }
0708                     }
0709                     else {moveable=true;}
0710                 }
0711             }
0712         }
0713     }
0714     else
0715     {
0716         xo=xo+1;
0717         if(xo==xn)
0718         {
0719             if(yn>yo)
0720             {
0721                 yo++;
0722                 if(yn==yo)
0723                 {
0724                     if(peice.isPeiceThere(xn, yn)==true)
0725                     {
0726                         StringTokenizer st = new StringTokenizer(
0727                             board[xn][yn]);
0728                         if(!playermoving.equals(st.nextToken()))
0729                         {

```

```

0729         moveable=true;
0730     }
0731 }
0732     else {moveable=true;}
0733 }
0734
0735 }
0736 if(yn<yo)
0737 {
0738     yo--;
0739     if(yn==yo)
0740     {
0741         if(peice.isPeiceThere(xn, yn)==true)
0742         {
0743             StringTokenizer st = new StringTokenizer(
0744                 board[xn][yn]);
0745             if(!playermoving.equals(st.nextToken()))
0746             {
0747                 moveable=true;
0748             }
0749         }
0750     }
0751     else {moveable=true;}
0752 }
0753 }
0754 }
0755 }
0756 else if(xn<xo)
0757 {
0758     xo--;
0759     if(xn==xo)
0760     {
0761         if(yo<yn)
0762         {
0763             yo=yo+2;
0764             if(yo==yn)
0765             {
0766                 if(peice.isPeiceThere(xn, yn)==true)
0767                 {
0768                     StringTokenizer st = new StringTokenizer(
0769                         board[xn][yn]);
0770                     if(!playermoving.equals(st.nextToken()))
0771                     {
0772                         moveable=true;
0773                     }
0774                 }
0775             }
0776         }
0777     }
0778     else if(yo>yn)
0779     {
0780         yo=yo-2;
0781         if(yo==yn)
0782         {
0783             if(peice.isPeiceThere(xn, yn)==true)
0784             {
0785                 StringTokenizer st = new StringTokenizer(
0786                     board[xn][yn]);
0787                 if(!playermoving.equals(st.nextToken()))
0788                 {
0789                     moveable=true;
0790                 }
0791             }

```

```

0792         else {moveable=true;}
0793     }
0794 }
0795 }
0796 else
0797 {
0798     xo=xo-1;
0799     if(xo==xn)
0800     {
0801         if(yo<yn)
0802         {
0803             yo++;
0804             if(yo==yn)
0805             {
0806                 if(peice.isPeiceThere(xn, yn)==true)
0807                 {
0808                     StringTokenizer st = new StringTokenizer(
0809                         board[xn][yn]);
0810                     if(!playermoving.equals(st.nextToken()))
0811                     {
0812                         moveable=true;
0813                     }
0814                 }
0815                 else {moveable=true;}
0816             }
0817         }
0818         else if(yo>yn)
0819         {
0820             yo--;
0821             if(yo==yn)
0822             {
0823                 if(peice.isPeiceThere(xn, yn)==true)
0824                 {
0825                     StringTokenizer st = new StringTokenizer(
0826                         board[xn][yn]);
0827                     if(!playermoving.equals(st.nextToken()))
0828                     {
0829                         moveable=true;
0830                     }
0831                 }
0832                 else {moveable=true;}
0833             }
0834         }
0835     }
0836 }
0837 }
0838
0839 return moveable;
0840 }
0841 /**
0842  * Checks if a king would be put into check by the move chosen by the
0843  * player.
0844  * @param peice
0845  * @param playermoving
0846  * @param xn
0847  * @param yn
0848  * @return true if he would be in check, false if he wouldn't
0849  */
0850 public boolean check(ChessMoves peice, // So the board and other
0851                     // methods within ChesMoves can be called.
0852                     String playerChecked, // The player that is moving their
0853                     // piece.
0854                     String pieceMoving, // The peice the player is moving

```

```

0855         // that my cause check.
0856         int xn,           // The location the piece is
0857         // moving to.
0858         int yn)         // same as above.
0859     {
0860         boolean inCheck=false;
0861         String currentBoard[][] = peice.outputBoard();
0862         int xKing = 0;
0863         int yKing = 0;
0864         String playerChecking = playerChecked;
0865         if(playerChecked.equals("P1")==true)
0866         {
0867             playerChecking="P2";
0868         }
0869         else if(playerChecked.equals("P2")==true)
0870         {
0871             playerChecking="P1";
0872         }
0873         for (int i=0;i<8;i++)
0874         {
0875             for (int j=0;j<8;j++)
0876             {
0877                 if(currentBoard[i][j]!=null)
0878                 {
0879                     if(currentBoard[i][j].equals(playerChecked + " " + pieceMoving))
0880                     {
0881                         currentBoard[xn][yn]=playerChecked + " " + pieceMoving;
0882                     }
0883                 }
0884             }
0885         }
0886         for (int i=0;i<8;i++)
0887         {
0888             for (int j=0;j<8;j++)
0889             {
0890                 if(currentBoard[i][j]!=null)
0891                 {
0892                     if(currentBoard[i][j].equals(playerChecked + " Ki"))
0893                     {
0894                         xKing=i;
0895                         yKing=j;
0896                     }
0897                 }
0898             }
0899         }
0900         for (int i=0;i<8;i++)
0901         {
0902             for (int j=0;j<8;j++)
0903             {
0904                 if(currentBoard[i][j]!=null)
0905                 {
0906                     StringTokenizer st = new StringTokenizer(
0907                         currentBoard[i][j]);
0908                     if(st.nextToken().equals(playerChecking))
0909                     {
0910                         String peiceToMove = st.nextToken();
0911                         if(peiceToMove.equals("R1")||
0912                             peiceToMove.equals("R2"))
0913                         {
0914                             inCheck = peice.possibleColumnMoves(
0915                                 peice, playerChecking, xKing, i, j);
0916                             if(inCheck!=true){inCheck =
0917                                 peice.possibleRowMoves(

```

```

0918         peice, playerChecking, yKing, j, i);
0919     }
0920     if(peiceToMove.equals("K1")||peiceToMove.equals("K2"))
0921     {
0922         inCheck = peice.possibleKnightMoves(
0923             peice, playerChecking, yKing, j, xKing, i);
0924     }
0925     if(peiceToMove.equals("B1")|| peiceToMove.equals("B2"))
0926     {
0927         inCheck = peice.possibleDiagonalMoves(
0928             peice, playerChecking, i, xKing, j, yKing);
0929     }
0930     if(peiceToMove.equals("Qu"))
0931     {
0932         inCheck = peice.possibleColumnMoves(
0933             peice, playerChecking, xKing, i, j);
0934         if(inCheck!=true){inCheck =
0935             peice.possibleRowMoves(
0936                 peice, playerChecking, yKing, j, i);
0937         if(inCheck!=true){inCheck =
0938             peice.possibleDiagonalMoves(
0939                 peice, playerChecking, i, xKing, i, yKing);}
0940     }
0941     }
0942     if(peiceToMove.equals("P1")|| peiceToMove.equals("P2")||
0943         peiceToMove.equals("P3")||peiceToMove.equals("P4")||
0944         peiceToMove.equals("P5")||peiceToMove.equals("P6")||
0945         peiceToMove.equals("P7")||peiceToMove.equals("P8"))
0946     {
0947         inCheck = peice.possiblePawnMoves(
0948             peice, playerChecking,
0949                 peiceToMove, yKing, j, xKing, i);
0950     }
0951     }
0952     }
0953     }
0954     }
0955     currentBoard[xn][yn]=null;
0956     return inCheck;
0957 }
0958 /**
0959  * Checks if the game should end because of checkmate.
0960  * @param peice
0961  * @param piece
0962  * @param playerChecking
0963  * @param xKing
0964  * @param yKing
0965  * @return true if the game will end, false if it will continue.
0966  */
0967 public boolean checkmate(ChessMoves peice, // So the board and other
0968     // methods within ChesMoves
0969     // can be called.
0970     String[][] piece, // The loactions of all of
0971     // the peices.
0972     String playerChecking, // The player putting the
0973     // king into checkmate.
0974     int xKing, // Location of the checked
0975     // king.
0976     int yKing) // same as above.
0977 {
0978     boolean endGame=false;
0979     String currentBoard[][] = piece;
0980     for (int i=0;i<8;i++)

```

```

0981     {
0982     for (int j=0;j<8;j++)
0983     {
0984         if(currentBoard[i][j]!=null)
0985         {
0986             if(i!=xKing&&j!=yKing)
0987             {
0988                 StringTokenizer st = new StringTokenizer(
0989                     currentBoard[i][j]);
0990                 if(st.nextToken().equals(playerChecking))
0991                 {
0992                     String peiceToMove = st.nextToken();
0993                     if(peiceToMove.equals("R1")||
0994                         peiceToMove.equals("R2"))
0995                     {
0996                         endGame = peice.possibleColumnMoves(peice,
0997                             playerChecking, xKing, i, j);
0998                         if(endGame!=true){endGame =
0999                             peice.possibleRowMoves(
1000                                 peice, playerChecking,
1001                                     yKing, j, i);}
1002                     }
1003                     if(peiceToMove.equals("K1")||
1004                         peiceToMove.equals("K2"))
1005                     {
1006                         endGame = peice.possibleKnightMoves(peice,
1007                             playerChecking, yKing, j, xKing, i);
1008                     }
1009                     if(peiceToMove.equals("B1")||
1010                         peiceToMove.equals("B2"))
1011                     {
1012                         endGame = peice.possibleDiagonalMoves(peice,
1013                             playerChecking, i, xKing, j, yKing);
1014                     }
1015                     if(peiceToMove.equals("Queen"))
1016                     {
1017                         endGame = peice.possibleColumnMoves(peice,
1018                             playerChecking, xKing, i, j);
1019                         if(endGame!=true)
1020                         {endGame = peice.possibleRowMoves(peice,
1021                             playerChecking, yKing, j, i);
1022                         if(endGame!=true)
1023                         {endGame = peice.possibleDiagonalMoves(peice,
1024                             playerChecking, i,
1025                                 xKing, i, yKing);}
1026                     }
1027                 }
1028                 if(peiceToMove.equals("P1")||
1029                     peiceToMove.equals("P2")||
1030                     peiceToMove.equals("P3")||
1031                     peiceToMove.equals("P4")||
1032                     peiceToMove.equals("P5")||
1033                     peiceToMove.equals("P6")||
1034                     peiceToMove.equals("P7")||
1035                     peiceToMove.equals("P8"))
1036                 {
1037                     endGame = peice.possiblePawnMoves(peice,
1038                         playerChecking, peiceToMove, yKing,
1039                             j, xKing, i);
1040                 }
1041             }
1042         }
1043     }

```

```

1044     }
1045     }
1046     return endGame;
1047     }
1048     String board[][];
1049
1050
1051 }
1052

/**=====
 * Class:      Stack      IB Dossier      Author: Mathew George
 * Version:    001                               Date: November 9, 2007
 *
 * Purpose     Contains the methods to build and implement a stack.
 *
 * Course:     Computer Science 33 IB           Teacher: Gerry Donaldson
 *
 * School:     Sir Winston Churchill High School, Calgary, Alberta, Canada
 *
 * Language:   Java J2SE 6.0 Target Operating System: Java Virtual Machine
 *
 * System:     Intel Celeron 3GHz running under Windows XP IDE: Eclipse 3.2
 *
=====*/

001 package src;
002
003 public class Stack
004 {
005     /**
006     * Constructs a new stack with space for the first item.
007     */
008     public Stack()
009     {
010         stack = new Object[1];
011         size = 1;
012     }
013     /**
014     * Adds items to the stack and will increase the size of the stack
015     * if it is already full.
016     * @param e
017     */
018     public void push(Object e) // The element being added to the Stack.
019     {
020         boolean eIn=false;
021         int i=0;
022         while(i<=size&&eIn==false)
023         {
024             if(isFull()==true)
025             {
026                 grow();
027                 i=0;
028             }
029             else if(stack[i]==null)
030             {
031                 stack[i]=e;
032                 eIn=true;
033             }
034             i++;
035         }

```

```

036 }
037 /**
038  * Returns the top element on the stack and removes it from the stack.
039  * @return returnValue
040  */
041 public Object pop()
042 {
043     Object returnValue = null;
044     boolean popped = false;
045     int i = 0;
046     if(size==1)
047     {
048         returnValue = stack[i];
049         stack[i]=null;
050         popped=true;
051     }
052     while(i<size&&popped==false)
053     {
054         if(stack[i]==null)
055         {
056             returnValue = stack[i-1];
057             stack[i-1]=null;
058             popped=true;
059         }
060         if(i==(size-1)&&popped==false)
061         {
062             if(stack[i]!=null)
063             {
064                 returnValue = stack[i];
065                 stack[i]=null;
066                 popped=true;
067             }
068         }
069         i++;
070     }
071
072     return returnValue;
073 }
074 /**
075  * Looks at the top value of the stack but does not remove it.
076  * @return returnValue
077  */
078 public Object peek()
079 {
080     Object returnValue = null;
081     int i = 0;
082     boolean peeked=false;
083     if(size==1)
084     {
085         returnValue = stack[i];
086         peeked=true;
087     }
088     else
089     {
090         while(i<size&&peeked==false)
091         {
092             if(stack[i]==null)
093             {
094                 returnValue = stack[i-1];
095                 peeked=true;
096             }
097             if(i==(size-1)&&peeked==false)
098             {

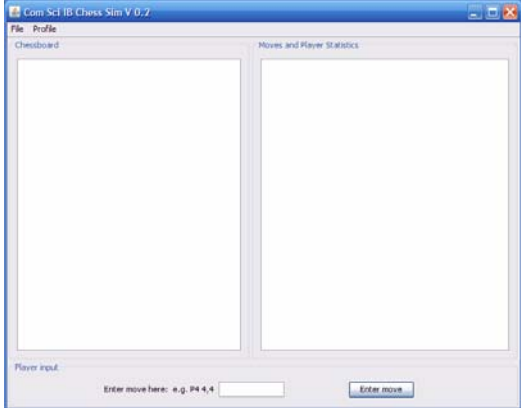
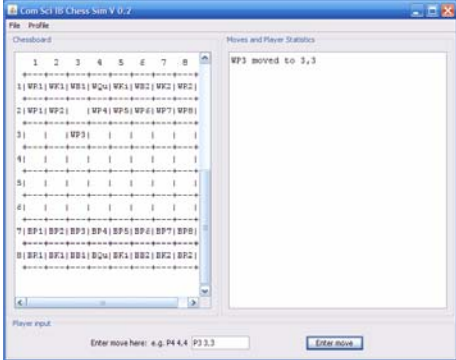
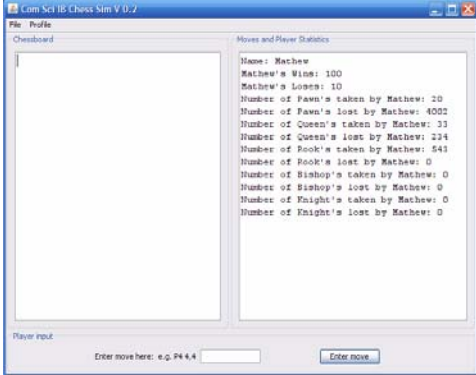
```

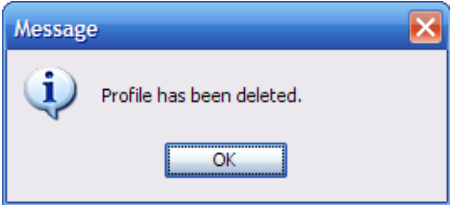
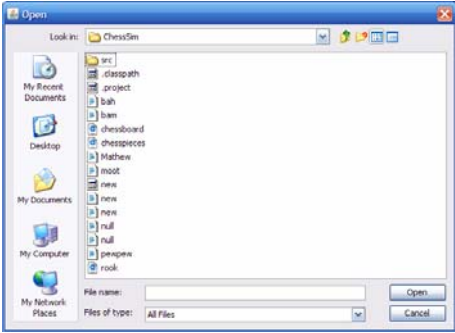
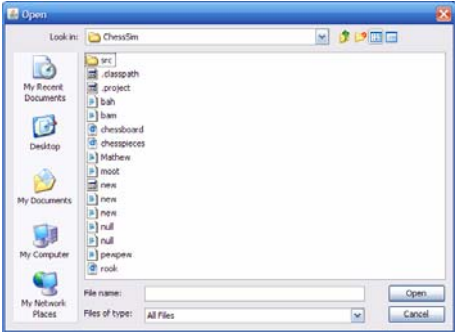
```

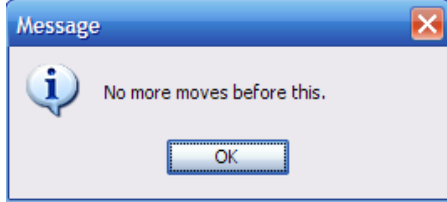
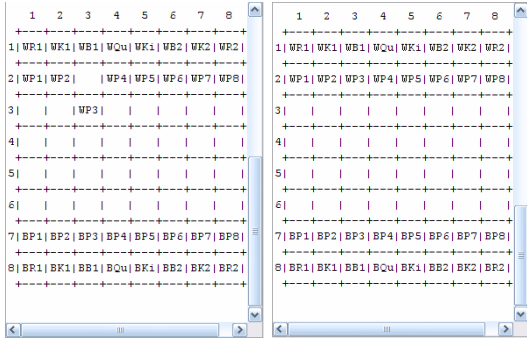
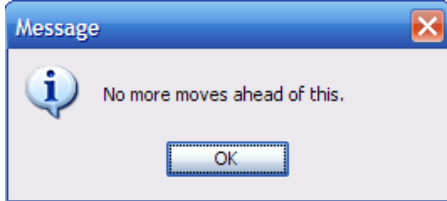
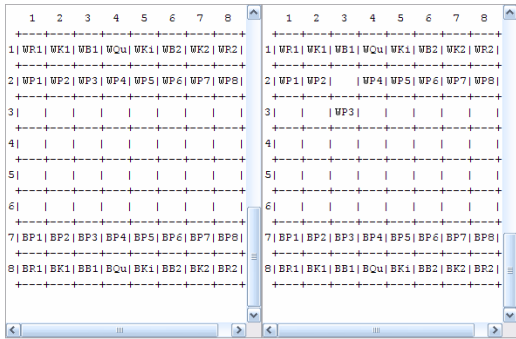
099         if(stack[i]!=null)
100         {
101             returnValue = stack[i];
102             peeked=true;
103         }
104     }
105     i++;
106 }
107 }
108 return returnValue;
109 }
110 /**
111  * Checks if the stack does not contain any elements.
112  * @return true if there aren't any, false if there are
113  */
114 public boolean isEmpty()
115 {
116     boolean empty = false;
117     if(stack[0]==null){empty=true;}
118     else{empty=false;}
119     return empty;
120 }
121 /**
122  * Checks if the stack does not contain any empty space.
123  * @return true if there are no null spaces, false if there are.
124  */
125 public boolean isFull()
126 {
127     boolean full = false;
128     for(int i=0;i<size;i++)
129     {
130         if(stack[i]!=null){full=true;}
131         else{full=false;}
132     }
133     return full;
134 }
135 public int size()
136 {
137     int actualSize =0;
138     for(int i=0;i<size;i++)
139     {
140         if(stack[i]!=null){actualSize++;}
141     }
142     return actualSize;
143 }
144 /**
145  * Doubles the space in the stack if it is full and a
146  * new element is trying to be added to the top of the stack.
147  */
148 private void grow()
149 {
150     int newSize = size+size;
151     size = newSize;
152     Object newStack[] = new Object[newSize];
153     System.arraycopy(stack, 0, newStack, 0, stack.length);
154     stack = newStack;
155 }
156 private Object stack[];
157 private int size;
158 }

```

C2: Usability

Objectives	Evidence that it is done
<p data-bbox="402 590 537 621">Uses a GUI</p>	 <p data-bbox="867 791 1263 819">Figure 13: the GUI for the chess simulator</p>
<p data-bbox="358 1024 618 1056">Multiple move outputs</p>	 <p data-bbox="867 1249 1211 1297">Figure 14: the GUI with two different Board outputs</p>
<p data-bbox="358 1564 630 1596">Player statistics output</p>	 <p data-bbox="867 1724 1211 1772">Figure 15: the GUI with two different Board outputs</p>

Objectives	Evidence that it is done
Profiles can be deleted	 <p data-bbox="878 764 1360 789">Figure 16: message that a profile has been deleted</p>
Games can be saved and loaded	 <p data-bbox="878 1167 1206 1192">Figure 17: Save/load file chooser</p>
Player profiles can be created and loaded	 <p data-bbox="865 1619 1320 1665">Figure 18: Create/load file chooser Note: both games and profiles are saved in the same place</p>

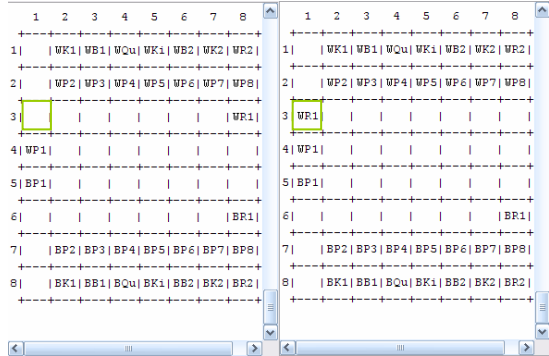
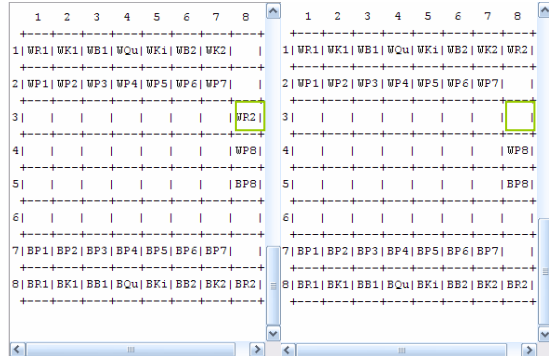
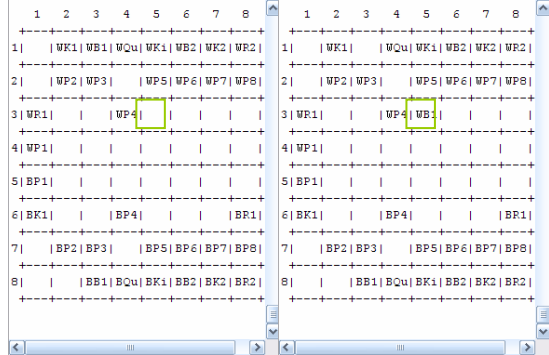
<p>Objectives</p>	<p>Evidence that it is done</p>
<p>Pasted moves can be loaded</p>	 <p>Figure 19: message that there are no previous moves</p>  <p>Figure 20: before/after shots of a move being retracted</p>
<p>After using the objective, a player can replace that move using another option</p>	 <p>Figure 21: message that there are no moves ahead of the current one</p>  <p>Figure 22: before/after shots of a move being redone</p>

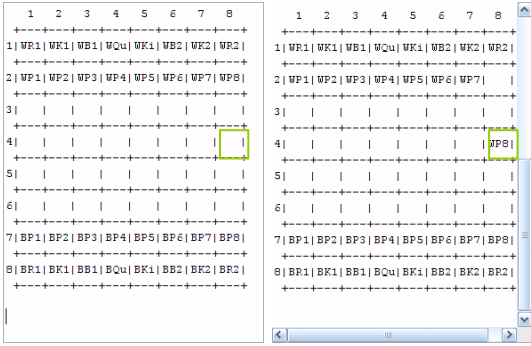
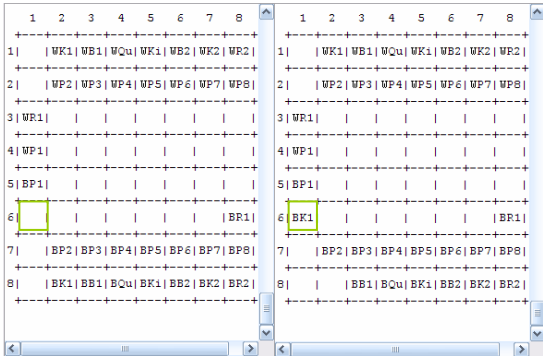
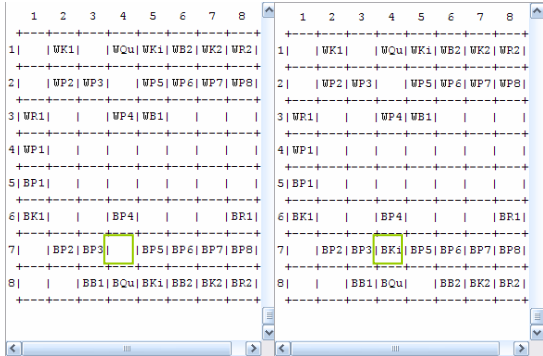
C3: Handling errors

Error	Handled
File not found exception	<code>JOptionPane.showMessageDialog(frame, "FileNotFoundException: " + e.getMessage());</code>
Piece does not exist	<code>JOptionPane.showMessageDialog(frame, "Piece does not exist" + "\n " + "(Note: piece names are CASE SENSITIVE)");</code> Is outputted if the piece does not exist even Covers the possibility of spelling mistakes.
File extension is wrong	<code>{JOptionPane.showMessageDialog(frame, "File Selected is not a profile.");}</code> Is outputted if the file selected in not a .pro file.
Profile is deleted	<code>JOptionPane.showMessageDialog(frame, "Profile has been deleted.");</code> Is outputted if the file selected counts as deleted (means the first token in the file is a 0)
IO Exception	<code>{ JOptionPane.showMessageDialog(frame, "IOException: " + e.getMessage());}</code> Is outputted if there is an IOException
Empty stack	<code>JOptionPane.showMessageDialog(frame, "No more moves before this.");</code> or this will be outputted <code>JOptionPane.showMessageDialog(frame, "No more moves ahead of this.");</code> If ether of the stacks is empty one of the above messages will be outputted.

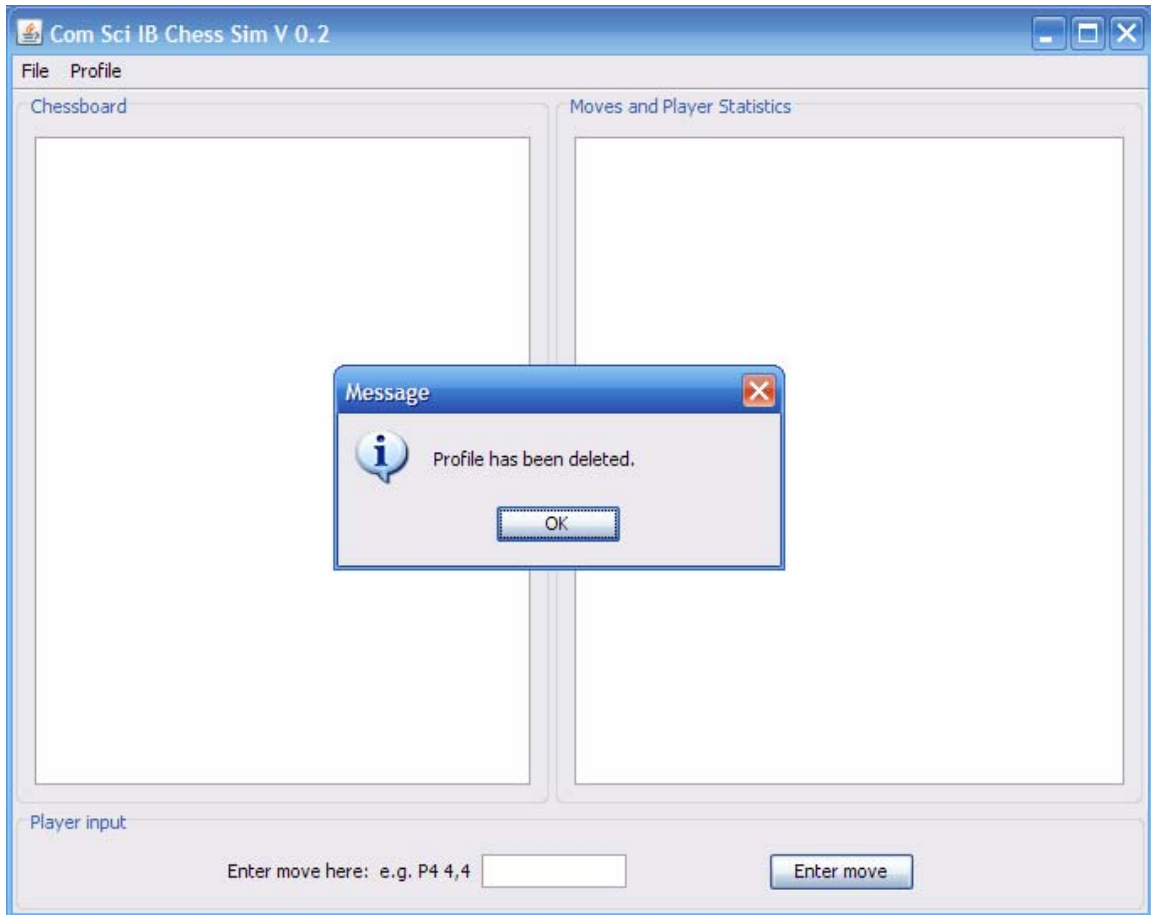
C4: Success of the Program

For the success of the GUI elements see section C2: Usability.

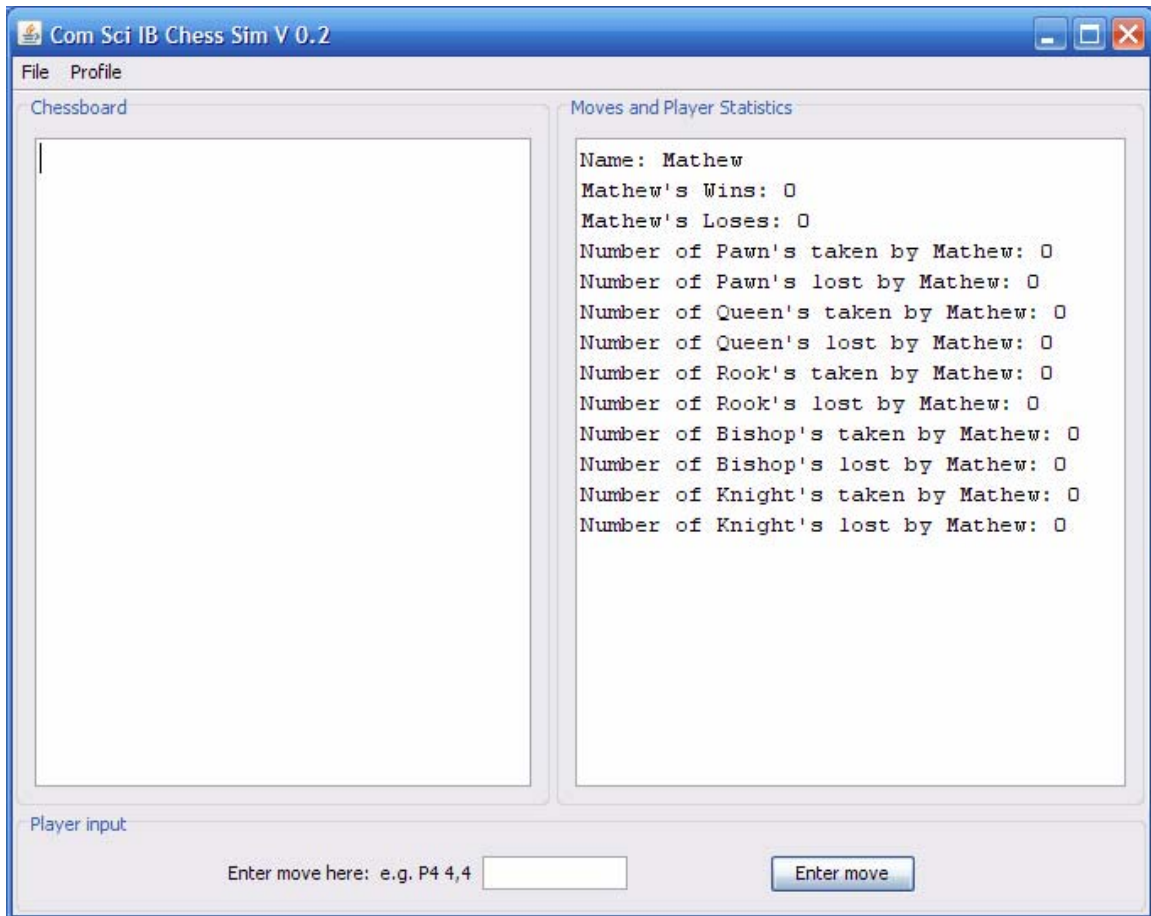
Objective	Evidence that it does
<p data-bbox="347 436 656 478">Horizontal move</p>	 <p data-bbox="829 709 1333 737">Figure 23: before/after shots of a horizontal move</p>
<p data-bbox="407 1003 553 1087">Vertical move</p>	 <p data-bbox="829 1218 1333 1245">Figure 24: before/after shots of a vertical move</p>
<p data-bbox="347 1539 630 1581">Diagonal move</p>	 <p data-bbox="829 1728 1333 1755">Figure 25: before/after shots of a diagonal move</p>

Objective	Evidence that it does
<p>Pawn move</p>	 <p>Figure 26: before/after shots of a pawn move</p>
<p>Knight move</p>	 <p>Figure 27: before/after shots of a knight move</p>
<p>King move</p>	 <p>Figure 28: before/after shots of a king move</p>

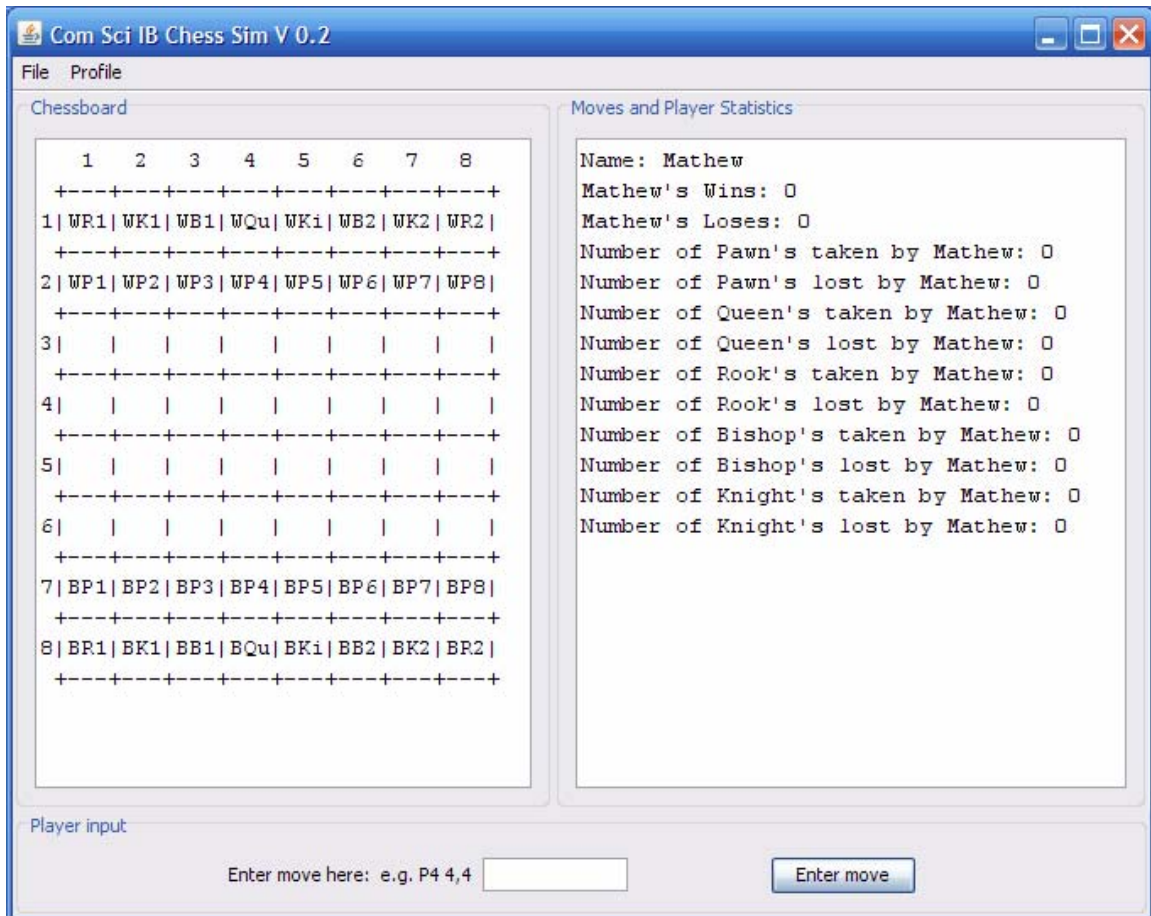
D1: Including an Annotated Hard Copy of the Test Output



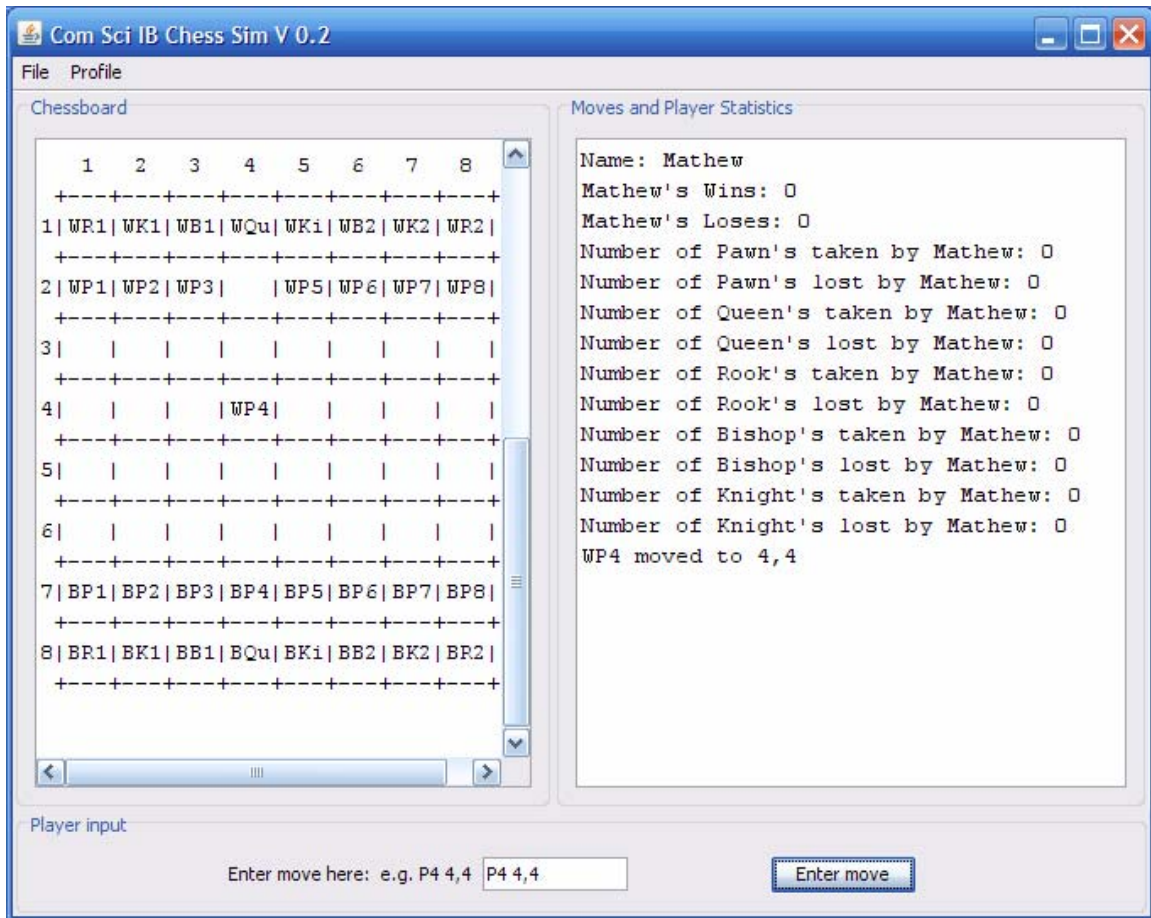
The above is an example of someone attempting to load a profile that has been deleted.



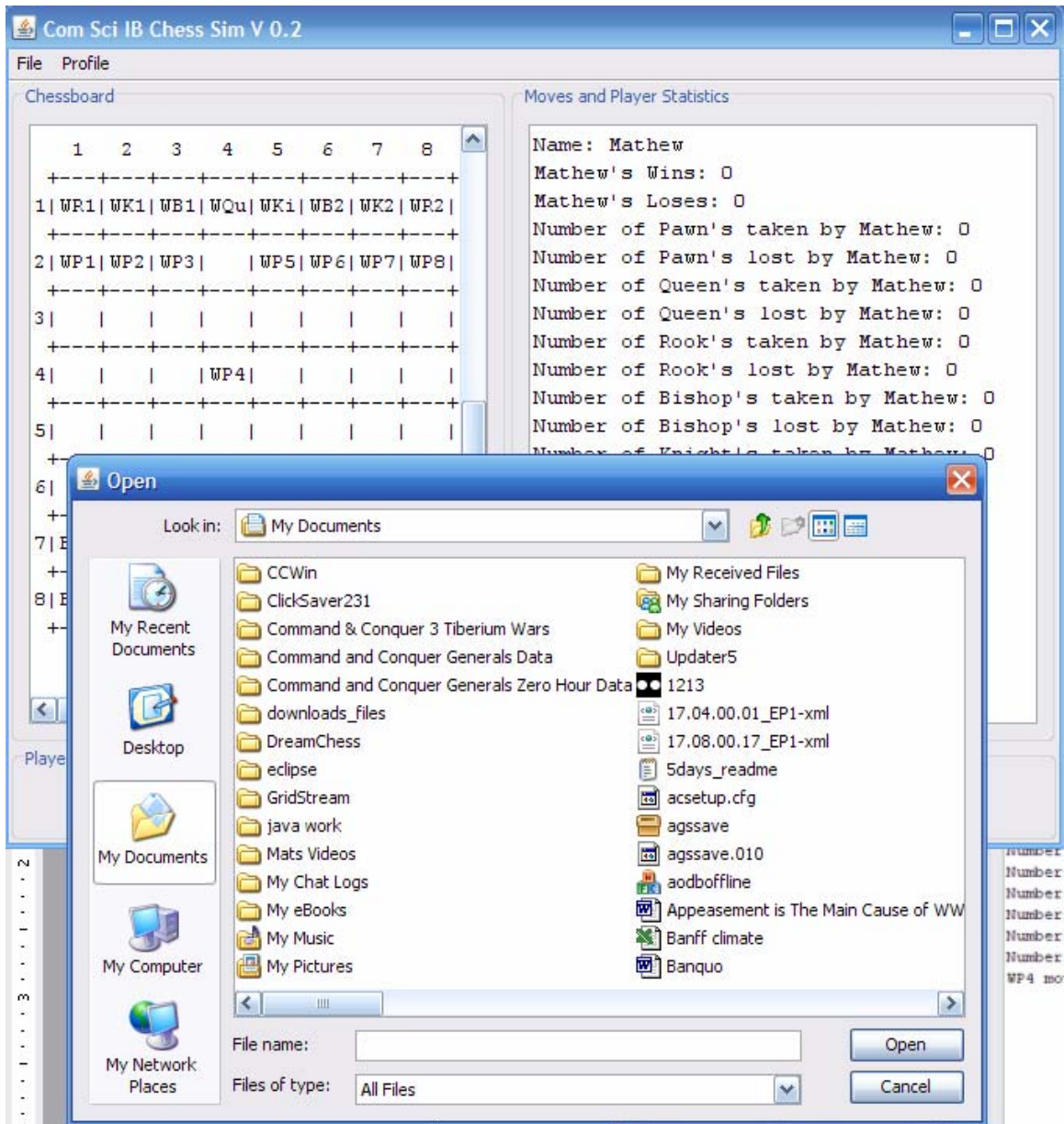
The above is an example of creating a new profile as you can see all of the data excluding the name is blank as it should be.



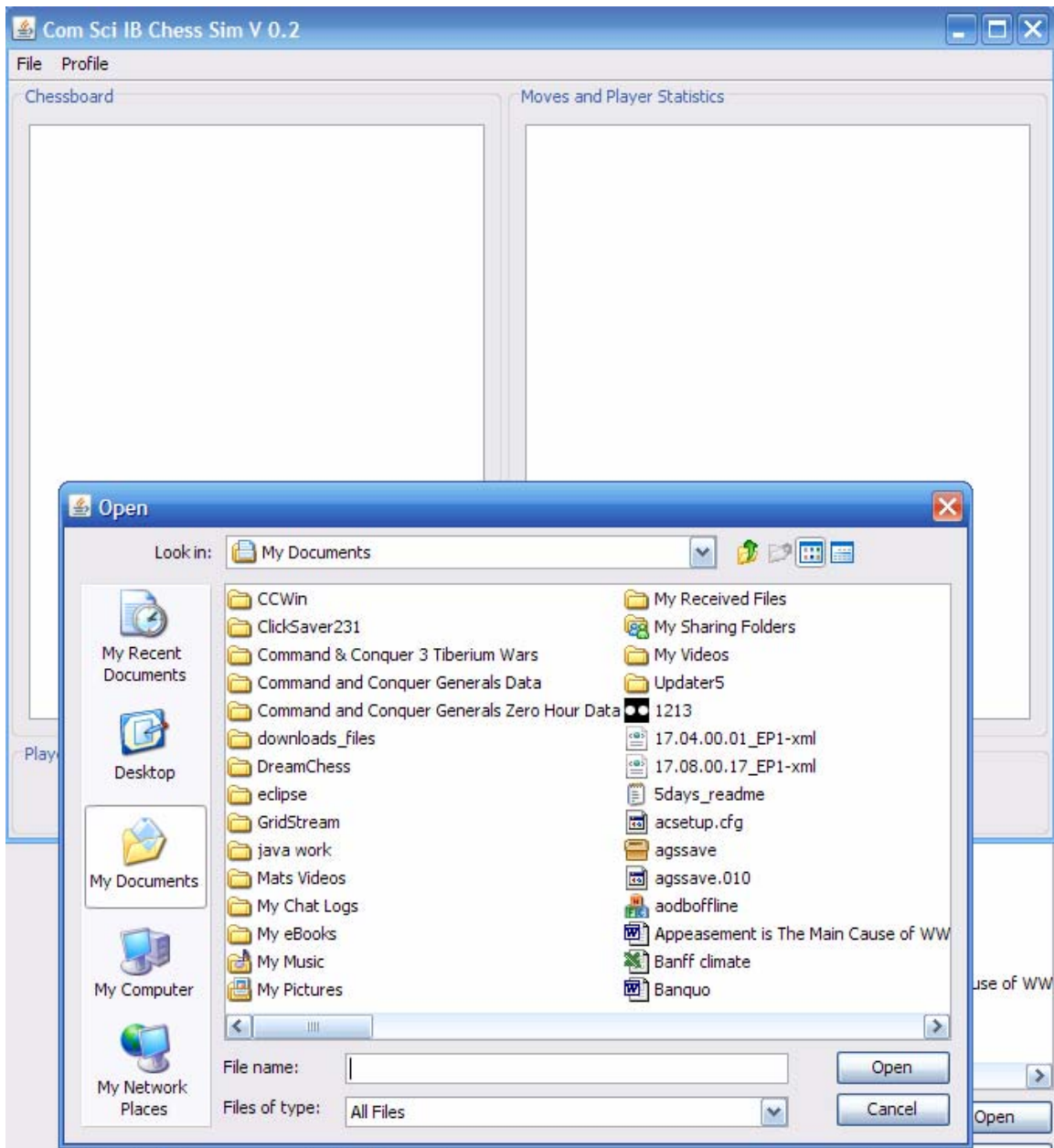
From the example above this one, the player has now started a game with only his new profile. It can be determined whether or not a second profile has been selected by making a move, since the data on the right is for player 1, and then attempting to get the statistics of the second player. If no output is shown, then there is no second player.



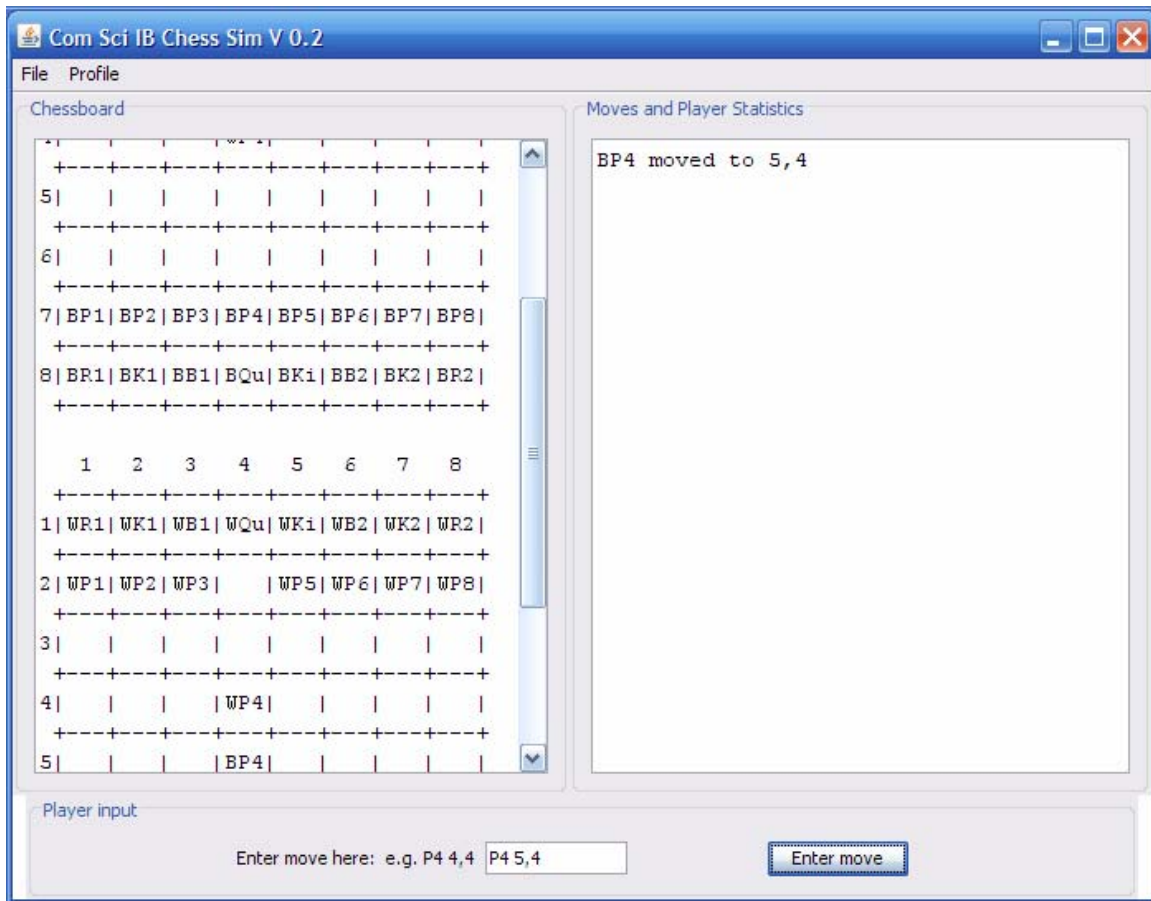
A move was made by player one, and then the player attempted to select there the statistics but no information was outputted therefore meaning that a second player does not have a profile.



The player is now attempting to save the current game. They would have to move to the folder where the java project is in order to save successfully. Due to the lack of file filters, there is excessive clutter in the folder it is currently in so this could be room for future improvement of the program. Note: players cannot replace old saves with new ones they must be deleted prior to re-saving.



After restarting the program, the player, who did not reload his profile, is attempting to load the game he had previously saved. The program does not require the old players to be using their profiles to continue again and it does not auto-load the profiles or stop two completely different people from continuing the game.



It may be difficult to see, but at the very top of the first board u can see that there is a piece on 4,4 and this is shown to be true if you look at the lower board and see that WP4 is there. On top of that, you can see that BP4 has been moved from 7,4 to 5,4 showing that the program does keep track of whose turn it was prior to saving as this is the same game as it was 2 outputs up.

D2: Evaluating Solutions

Did it Work?

While I did not show any shots of the program acting buggy, though exasutive testing, I had found several inexplicable errors kept rearing their ugly heads. The most perplexing of them was an error involving the horizontal movement script. I would move R2 down to 3,8 on the board then attempt to move it across to 3,1. After several attempts with the only output being Cannont be moved there, I attempted to move it anywhere on that row. All attempts ended in the same way, Cannont be moved there. After this I thought maybe there was a counting error and tried a value in the row above it, I attempted to move to 4,1 and oddly enough it worked. However, this is not the odd thing about it, after this I wanted to see if the movement in the other direction was working, so I could compare the two scripts and try and purge the error that way. To my surprise, not only did it work when R1 was moved from 1,3 to 3,8 but it worked fine in the other direction! After scratching my head for a couple of hours and with no reasonable explanation for this error I moved on because it was clear that no matter what I would do, this error would not be fixed. Aside from several minor errors such as these ones, the program worked fine though.

Did it address the criteria for success and any limitations?

The first thing on the list of things I must consider for the program to be successful is if the players can create and access previously made profiles. I have been able to inprove on the original idea by allowing players to login at anytime to a profile. However, the players are still only able to log out by quitting the game making it so

people can not log in and out only if it is beneficial for them. Also, there is no password protection on profiles which many users would defiantly like to have.

The next thing, even prior to making the game mechanics, I set-up was the save and load game mechanics. When saving, it saves all the data in one row with the first value being whose turn it is so the game makes sure that the turn order is conserved. However, the program does not take into account previous moves from the game before it was saved if you attempt to load it.

This was a simple addition to the game that I added immediately after the preliminary game mechanics were finished as it was an easy way to test if the game mechanics were working as intended. This part is however limited in that a major error can easily go by unnoticed unless the user is watching very carefully. Also little in the way of feed back is given to the user and if errors start cropping up, they may not be able to determine exactly what is wrong.

Once most of the noticeable errors in the game mechanics were cleared up, I moved on to this, and the last major addition to the project, the ability to move to pasted and future moves, if they exist. This was simply done by making two separate stacks. Each time a move was made, the first stack would be filled and everytime a move in the first stack was recalled, the current move would be stored in the second stack. On top of this, the second stack, future moves, would be reset if the players decide to continue from a previous point in the game. The system is limited because it does not remove statistics added to players if it passes over a move where a piece was taken. Also, as stated in the save game section above, they are not stored with the saved games.

Right after I made the profile system, which was at the very start, I added this system since it was, like the visual board section, a good way to test if the program was loading the information from the files properly. It simply outputted all the information in the playerStats arrays in a pre-ordered fashion to one of the text output areas. The only way I can think of to improve this system would be to add statistics but much more would be stretched for usefulness.

What additional features could the program have?

A major feature upgrade would be to change things to true graphics as opposed to text based information on GUI systems. Also, I stated this one in the section above, adding password protection to profiles for both account safety and to make users feel more secure that their information is not being altered without the consent. Though the next options should have been add, many chess simulators do not offer them as an option, the addition of castling and pawn promotion would be an almost necessary edition to future versions of the chess simulator.

Was the initial design appropriate?

The initial design was almost perfect for what I needed. There were some minor changes necessary because the coding would be easier the way I have them now, but overall the initial design. My first plan to have full graphics for the board was a bit ambitious since my skill with drawing graphics, both by hand and using Java, are not very good making it difficult, if not impossible to use full graphics as opposed to text based programs. However, if I was able to get the graphics working, it would be a considerable jump in the quality of the program since it will increase both the presentation and the user-friendliness.

D3: Including User Documentation

Running:

A computer running this program will require at minimum:

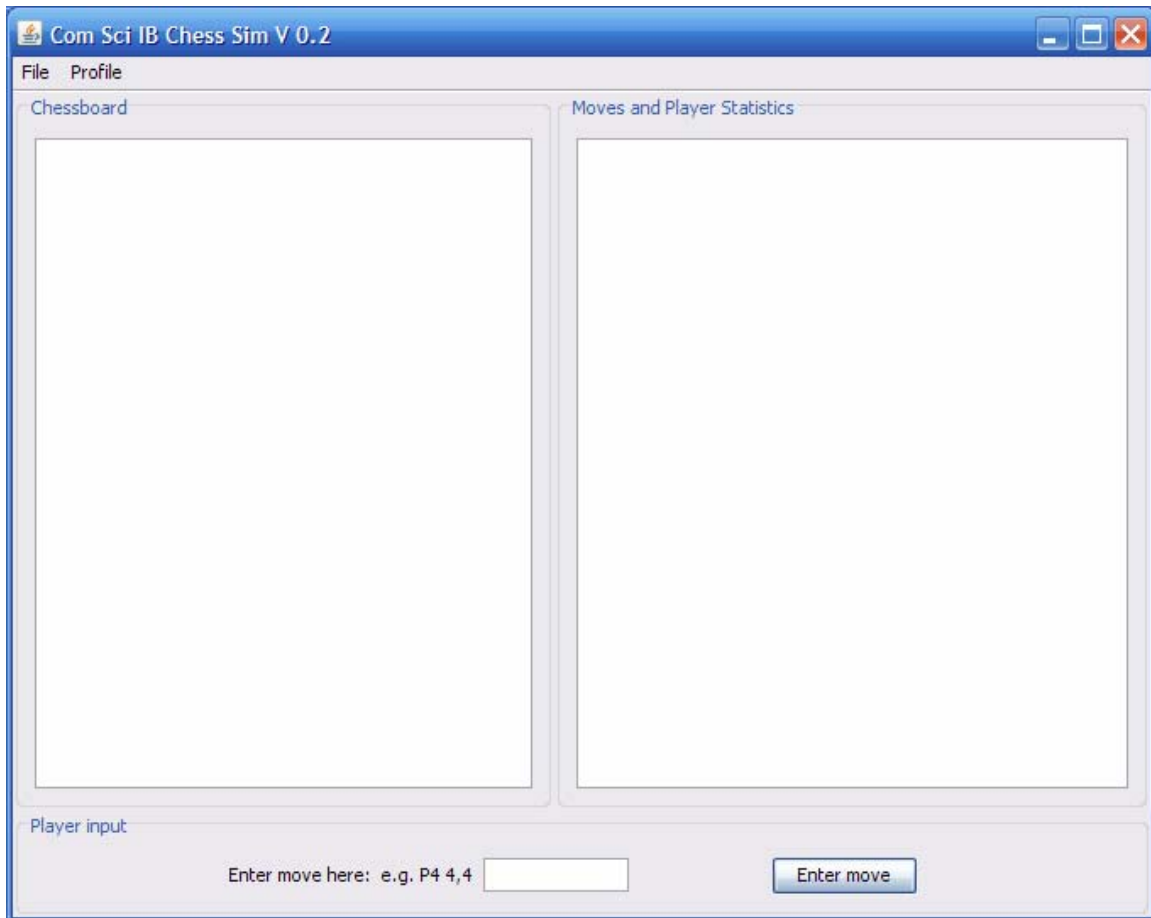
- Java J2SE 6.0
- Operating System: Java Virtual Machine

Note: future versions of java maybe be incompatible with the program but this is unlikely.

If your computer does meet the minimum standards then go to the java class Chessboard and run it. The easiest way to accomplish this would be to download the third-party application eclipse from <http://www.eclipse.org/> and select the download option. The program does not require installation to run just select the .exe file with then import the ChessSim project. After importing the project, simply select the run button from the top it is the 5th button over and looks like a green circle with a white arrow pointing forward.

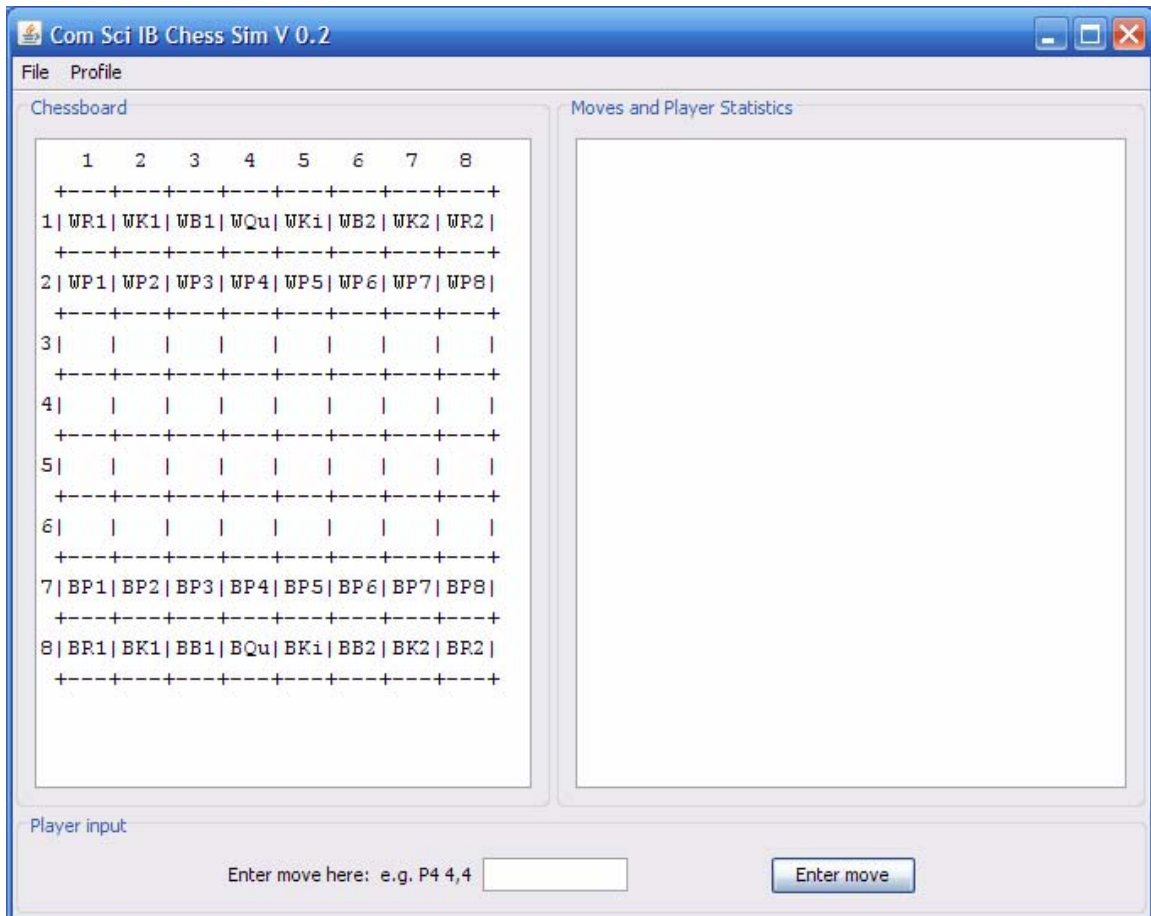
Using the Program:

Now that the program is running, you will see this:

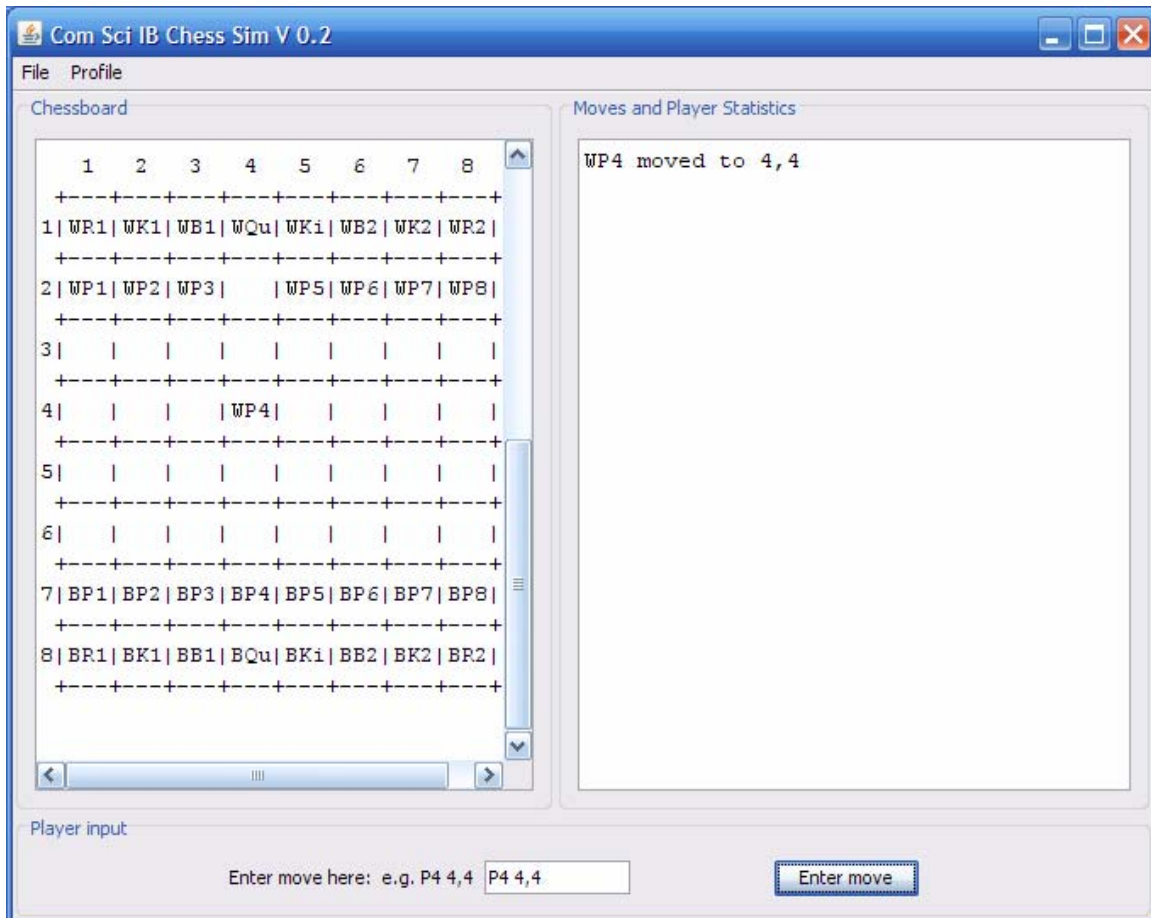


To create a new profile with the game select File->New Profile. Once the File Chooser pops up, navigate your way to the project file, the file labeled ChessSim, and save your file as the name you want and “.pro” at the end of it. If you forget the “.pro” the file will not run and you will have to try again. Note: names cannot have more than 24 characters in their name.

Now that you have a profile, You can start a game by going back into the file menu and selecting “New Game” immediately afterward a board will appear in the text box on the left like so:



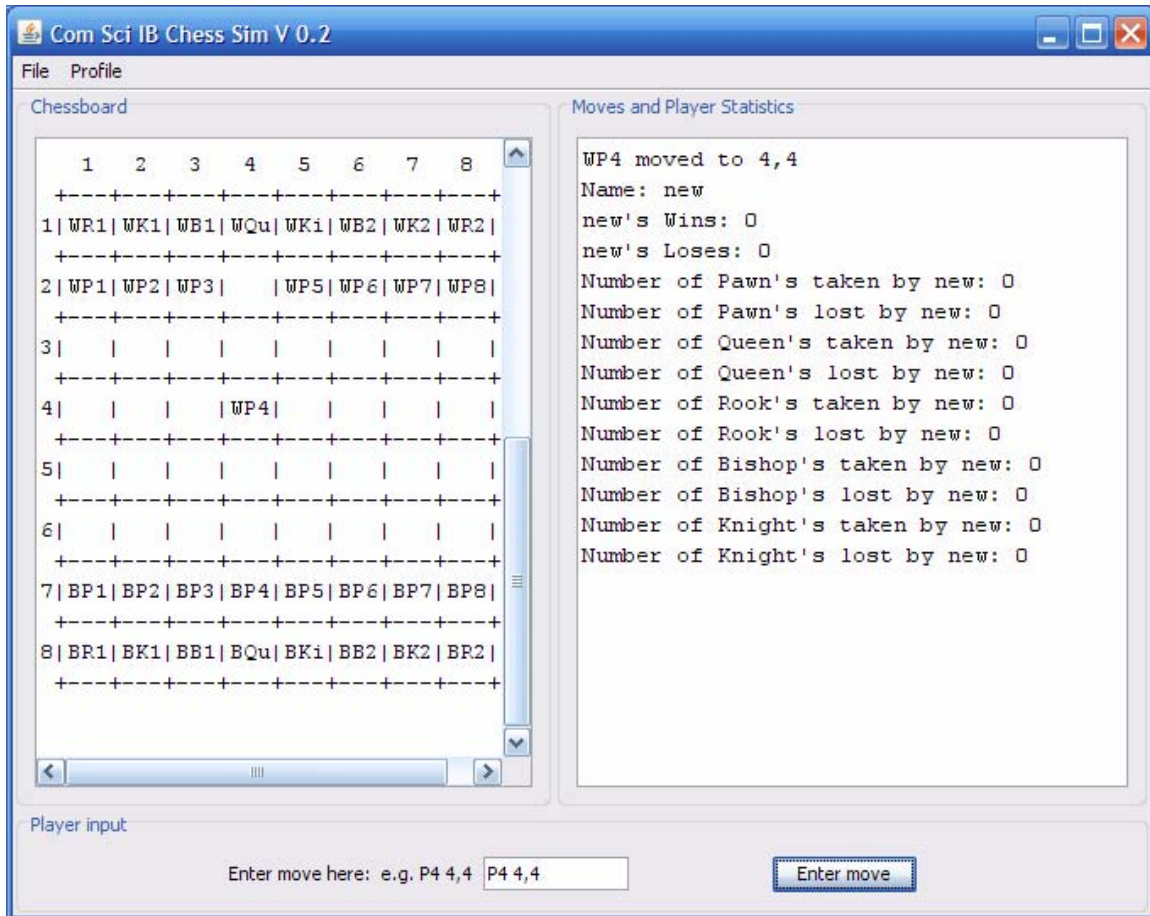
Now that a game has been started, moves can be made by entering the piece, without side denominations such as “W” or “B”, followed by the coordinates of the square u want to move it to, row first then column. The game will tall you if it’s a legal move by outputting a new board and posting the move in the right hand box.



From here you can save this game or load a new game by selecting those options from the File menu. A File Chooser similar to the one for Create Profile will pop up and you have to follow the same procedure except there is no cap on the number of characters the name can be and the name must end in “.gam” otherwise the program will be unable to load the game.

Also, since a move has been made the last move option can now be used. This option enables the user to go back as many moves as there are in the game and return to the current move. Note: if you decide to continue forward though a different path of moves, your old moves ahead of that one will be lost.

If you want to see how well you are doing go up to the Profile menu and select Stats. You will get an output like:



Note: it must be your turn to output your statistics and the first player to login is auto-set to player 1 slot.

The final option is to delete a profile. The profile will still be listed in the directory and in fact still exists. However, the program will be unable to retrieve the data within it the file so make sure you are sure you want to get rid of the data before you hit the delete button.

Exiting the Program:

To exit the program simply press the red X button in the upper right corner. Note: any unsaved data will be lost forever the game only saves accounts when a game is saved or finished.

E: Holistic Approach to the Dossier

I was very attentive to helping others when they were in need of suggestions for how to improve their programs. For instance, a classmate of mine Jory Stewart, was looking for a way to make sure that one of his programs mechanics was pseudo-random so I suggested that he use the random number generator class and set it to between 1 and 100 enabling him to set only certain percent chances certain things working i.e. a 1% chance item would only work if say 1 was pulled up by the random number generator.

I have been working consistently on the programming of my dossier for two weeks straight working for about 4-5 hours a day until I was finished with the programming then spend half a week working 6-7 hours a day getting the written work done on my dossier to be handed in on time.

Unfortunately I was not able to complete everything I set out to do. In addition to the work I have finished I had hoped to introduced different backgrounds to make it easier for the users to tell whose turn it is but sadly had to be scrapped when I could think of no practical way to implement this into my program. On top of this I missed several deadlines but was still more than willing to show Mr. Donaldson, my computer science teacher, what I had accomplished to show that I was still making fair headway on the project, just not as much as he had hoped.