Chess for students; written in Python

Alexander Saoutkin

Candidate Number: 2159

Centre Number: 10834

March 2017

Contents

List of Figures

List of Tables

Analysis

1.1 Introduction

The goal of this project is to build a new system for a teacher who runs a chess club At Alleyn's School, London. In this section, research has been done to learn what the current system does. This research will inform me of what the new system that I will build should be able to do and what additional features it will provide over the current implementation. My research consisted of giving a questionnaire to the user and of researching the rules of chess.

1.2 Rules of Chess

As with any game, there are rules that have to be adhered to for a valid game to be played. For me to have a viable solution to the problem identified, the program I create must follow the rules of chess. Thus in this section I will discuss what these rules are.

1.2.1   The Game Board

Figure 1.1: The initial setup of a game of chess.

The initial starting position of the board contains all the types of pieces in the game: pawn, king, queen, bishop, knight and rook. From this point onwards, white is the first to move. Once white has completed a legal move it is then black's turn to do the same and the game continues until the game has reached the state of checkmate and stalemate, which will be explained in detail later. Only one piece can occupy a square on the board, thus a piece cannot take one of its own pieces but can capture opposing pieces as long as it isn't a king.

Note: when the types of moves that a certain type of piece can make are discussed below, it is assumed that when moved, it does not put the player who is moving the piece into check. In addition, it will be assumed that the move does not mean that the moving piece is in the same place. If this is the case then the move is illegal.

2

1.2.2   Pawns

Figure 1.2: The possible moves a pawn can make.

The pawn is a type of piece and has two types of moves, depending on the situation. If the pawn is in the same space as its starting position then it can either move one or two spaces forward, given that there is no piece in the way of this movement. In any other position it is only able to move into the next square forward, as long as there is no other piece currently there.

Promotion
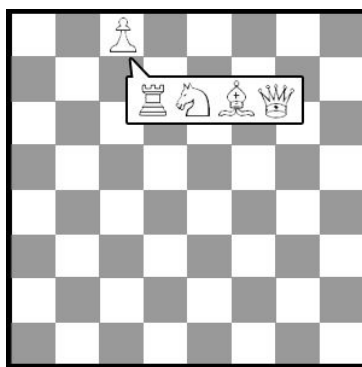
Figure 1.3: The types of pieces that a pawn can be promoted to.

When a pawn reaches the other side of the board the pawn must be changed to an additional piece that is a rook, knight, bishop or queen.

En Passant

Figure 1.4: A board position in which en passant is possible.

As can be seen above, if a pawn is moved two spaces forward such that it is horizontally adjacent to an opposing pawn once it is moved, the opposing pawn has an opportunity to capture the pawn and move to the x' marked on the diagram. When this opportunity occurs, whether it is taken or not, it cannot be done again by that player for the rest of the game.

1.2.3   Kings
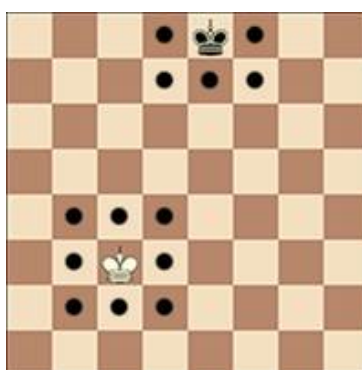
Figure 1.5: The possible moves of a king.

The king can move to any adjacent square provided that it is not in check once it completes the move. In addition it should not be adjacent diagonally, horizontally or vertically, to the opposing king after it has moved.

Castling
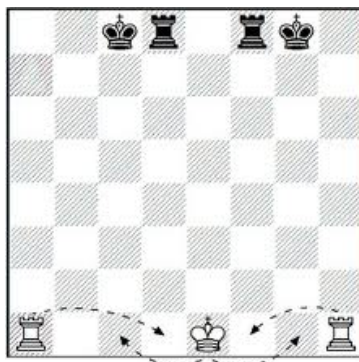
Figure 1.6: Movement of king and rook during castling.

Castling is a move (shown above) that can be performed only if the king and rook have not been moved before. For castling to be performed no pieces must be in the way of the king and the rook moving and any of the squares in between them must not be attacked by an opposing piece.

1.2.4   Bishop

Figure 1.7: The possible moves of a bishop.

The bishop can move diagonally in any direction. Thus, it can only move on the colour of squares that are the same as its starting position.

1.2.5   Rook

Figure 1.8: The possibles moves of a rook.

The rook can move both vertically and horizontally in any direction.

1.2.6   Queen
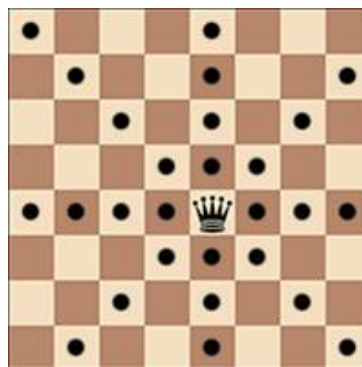
Figure 1.9: The possible moves of a queen.

The queen can both move anywhere diagonally, horizontally and vertically. In other words, a queen can be considered a combination of both the rook and the bishop.

1.2.7   Knight

Figure 1.10: The possible moves of a knight.

The knight has one of the more interesting moving patterns. It can move in what is known as an L-shape. To be more precise, the possible moves of a knight can be calculated by moving 2 along vertically and then 1 horizontally either side, and vice-versa. Its attacking moves do not include the path to its possible moves.

1.2.8   Game States

In chess there are 3 types of game states that differ from normal play and restrict what legal moves can be made. Two of those are permanent (end of game) states: checkmate and stalemate. The other is check and this means that only moves that take a player out of check can be made.

Check
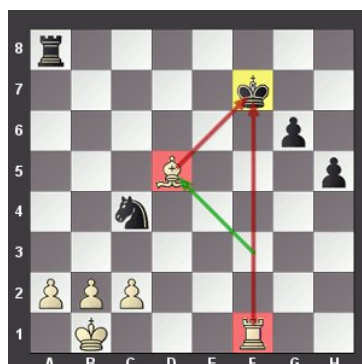
Figure 1.11: An example of check.

Check occurs when a piece moves such that it is attacking the opposing king. An attacking move is such that if we theoretically suppose that the king is a piece that can be captured by a piece, then it would be captured if the piece moved there. An example of this is the image seen above. An important note is that for pawns this means they cannot check a king if it is horizontally ahead of it, as a pawn cannot move there if we theoretically supposed that a king is able to be captured. However, it can if the king is diagonally adjacent in front of the pawn.

Checkmate
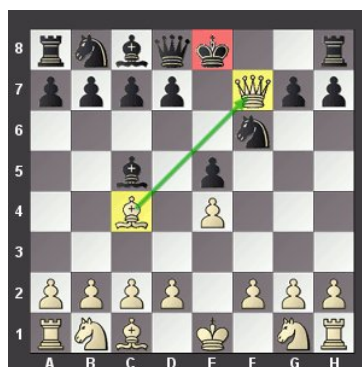
Figure 1.12: An example of checkmate.

Checkmate occurs when a player is put into check and cannot make a move that brings themselves out of check. The person who is in check is the loser of the game and thus the opposing player is the winner. A well-known example is the scholar's mate which can be seen above. The queen cannot be taken by any piece and the king has nowhere to move without still being in check. In addition, the king cannot take the queen as it will put itself in check to the bishop. Thus black concedes the game as it is in a state of checkmate.

Stalemate
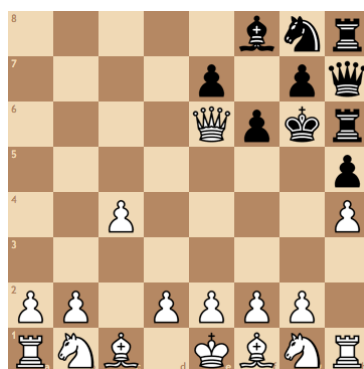
Figure 1.13: An example of stalemate.

Stalemate occurs when a player who is not in check cannot make any legal moves and thus the game cannot continue. Above is an example of a stalemate. It is black's turn to move but it has no legal moves to make. Any movements are either not available to make or expose the king to being in check to either the far-right pawn or the queen. When a stalemate occurs the game is declared as drawn and there are no winners or losers.

1.3   Questionnaire

As mentioned in section 1.1, a questionnaire would form part of my research. The teacher who runs the chess club is the one who has answered the following questionnaire. In this

questionnaire, it is determined what the current system does, the problems it has, and the objectives of the new system which I will be creating.

Explain the background of the current system.

I work at Alleyn's School in Dulwich as a teacher and run the chess club every Wednesday lunch. Pupils from ages 11-18 are allowed to play chess. Every year we run an inter-house chess competition, where each house of the school assembles a team together to determine which house is best at chess. Approximately 20 pupils show up to the club every week, with this number rising to about 50 during competitions. The inter-house chess competition is particularly popular and I run it once a year for about 4 or 5 weeks. The competition take place in two rooms in the Science Department. The competition is played on approximately 25 Chess Boards.

What kind of data is stored on the system and how is it processed?

When the chess competition happens, we must store data on the results of individual games. Specifically, we store the names of the player and the winner of each game. The game board used by the players serves as a way of recording the state of the game. The game board also serves as a way of making moves and seeing moves being made. The rules of the game are adhered to by the players themselves, as it is assumed that is in the best interest of each player to make sure that the other is not cheating. If a game overruns past lunchtime, then the players can take a picture of the board and then continue the game later. This information is currently displayed on the whiteboard during the lunchtime of the competition, and then is transferred to my personal notebook once lunchtime is over.

What problems does this current system have?

Firstly, there can be times where there are disagreements between two players over the game. The issues tend to be centred around possible cheating. It would be good to have a system where arguments over cheating would not occur. Secondly, there are many clubs and activities run at the school, which clash with chess club. This results in people who cannot play chess with us even though they want to. It would be nice to be able

to allow people to play chess outside of school hours. Thirdly, whilst a photo of the game board can be taken to allow the game to be continued elsewhere if it runs over lunch, information about whether en passant is still possible and whos move it is cannot be done via a picture. In addition, when a chess competition occurs, many players are interested in seeing the results of certain games. However, as all games are stored on my notebook students are usually individually told or e-mailed the results of games if they are interested. It would be useful to have a system in place where interested parties can view the results of all chess games played.

What data would you like to be stored in a new system?

I would like for games to be stored. The data on each game should be such that it is possible to continue playing later. Thus, it must include the position of all the pieces on the board, the current game state (e.g. is a player in check?), whether en passant is possible and where it can occur, and whose turn it is. In addition, the names of both players must be stored, the number of moves made, when the last time the game was played and the name of the winner if there is one.

Could you outline the core requirements of the new system?

The new system should be able to identify all the legal moves of a given piece and display this information clearly to the user. In addition, the game must be able to identify changes in the state of the game and display these changes to the user when they occur. These requirements must be satisfied as it should be possible to play chess using this new system, where moves can be made and the results of these moves should be clearly shown. All games that are played must have the ability to be saved, including the names of the players, so that the game can be played at a later date. Thus, all games that are saved must be able to be easily loaded by the user.

1.4   Setting

The program was designed for the school's Chess club and is open for students from year 7 to 13. Chess club currently runs weekly at lunchtimes and is run by a physics teacher Dr De Silva. Every year a primarily student run inter-house chess competition, with each house having a team of avid chess players, takes place to determine which house has the best chess players. As a member of chess club I identified and researched problems with the current system that the club uses. I also consulted with Dr De Silva about any problems that he has noted in the current system and have used this information to aid me in determining the objectives that I plan for the new system to implement.

1.5   Users

The primary users are those students who partake in the chess club and actively play chess amongst one another. It can also be used by the teacher that runs the club to record the results of games, which is paramount when there are competitions taking place. There is an additional benefit in that it can be played at any time and on any computer in the school area that has the program installed. This may be of good use to those who would like to play chess but are unable to go to the club due to other extra-curricular activities.

1.6   System

Currently most games that are played are not recorded because most are for recreational entertainment and so the result of the game is inconsequential, though results may be remembered by students and also may help in choosing players for the inter-house chess competition. Currently, when the inter-house chess competition happens the whiteboard in the classroom is used to show players who they are playing against and when the game is finished the result is written to the right of the matchings. Then the results are

12

transferred to paper and are stored by the teacher who runs the club.

1.7   Input

All the results of games to do with a competition are recorded in a notebook and of course the movement of pieces is recorded on the board. All other games are not recorded at all and the location of pieces when a game is finished are generally not stored unless a student chooses to take a picture of the board.

1.8   Processing

Processing in the current system is fairly simple. Once a player has chosen where to move a piece, they pick it up and move it to the appropriate position on the board. If it is illegal it is assumed that the opponent will notice as it is in their best interest to not allow such a move. If that is the case then the piece is simply moved back to its place. The player who moves a piece also notes mentally if the piece that they have moved has resulted in a change in the state of the game.

1.9   Output

Output is simply where the player moves their piece on the board, which is visible to both players and any people watching. In addition, when a player moves a piece which results in the changing of the game state they must state this change openly to the opposing player.

1.10   Volumetrics

In this particular application it is hard to estimate the memory needed to store the data needed to play a game as the data required is dynamic. However, we can make some assumptions about the game and use this to estimate the memory required.

Firstly, I'll discuss the byte allocations that the Python programming language gives to certain objects and data structures:

Bytes	Type	Details
24	int	
24	bool	an instance of an int
52	str (unicode)	+4 byte per additional character.
72	list	+32 for first element, 8 thereafter.
280	dict	6th item increases to 1048; 22nd, 3352; 86th, 12568 *

Table 1.1: Byte allocations for primitive data types in Python.

Firstly, the game is stored in a 2D (8×8) list object, with each element of the list containing either an integer with value 0 or a Piece object. However, this is not how the data is stored in the JSON file. Instead, the JSON file mimics all the attributes of the `class Board(object)`, apart from the `board` attribute.

In the following table, a list of all the attributes of the board class are contained with the type that they will be stored as in the JSON file. This table will show how many bytes are required to store a game as an upper bound. It is assumed that the player names are 10 characters long, which is unlikely to be the case for one player, let alone two. In addition, it is assumed that no pieces are taken, as if that is the case, then the size of the file will inevitably be smaller.

14

Attribute	Type	Bytes	Occurence
id	int	24	1
player_one	str	$52 + 4 \times 8 = 84$ (assuming 10 characters).	1
player_two	str	$52 + 4 \times 8 = 84$ (assuming 10 characters).	1
last_played	str	$52 + 4 \times 10 = 92$ (in format dd/mm/yyyy)	1
turn	str	$52 + 4 \times 5 = 72$ ("White" or "Black")	1
move_num	int	24	1
winner	str	$52 + 4 \times 5 = 72$ ("White" or "Black")	1
colour_in_check	str	$52 + 4 \times 5 = 72$ ("White" or "Black")	1
is_stalemate	bool(int)	24	1
game_over	bool(int)	24	1
must_promote	bool(int)	24	1
enpassant_possible	dict	280	1
enpassant_possible['Black']	bool(int)	24	1
enpassant_possible['White']	bool(int)	24	1
enpassant_move	dict	280	1
enpassant_move['from']	list	$72 + 32 + 24 + 8 + 2 = 150$	1
enpassant_move['to']	list	$72 + 32 + 24 + 8 + 24 = 150$	1
enpassant_move['taken']	list	$72 + 32 + 24 + 8 + 24 = 150$	1
pieces	dict	280	1
pieces[*]	dict	280	5
pieces[*]['position']	list	$72 + 32 + 24 + 8 + 24 = 150$	32 (all pieces)
pieces[*]['colour']	str	$52 + 4 \times 5 = 72$ ("White" or "Black")	32 (all pieces)
pieces[*]['first_moved']	int	24	16 (all pawns)
pieces[*]['has_moved']	list	$72 + 32 + 24 + 8 + 24 = 150$	6 (rooks and kings)
Total		11,722 bytes	

Table 1.2: Bytes required for the storing of one chess game.

Whilst the following table helps to calculate the size of storing each game, it is not taken into account the overhead of the game list. Thus the formula, $f(x)$, where x is the number of games stored, for the upper bound in bytes of the JSON file is:

$$f(x) = (11722)x + (72 + 32) + 8(x - 1) = 11730x + 96$$

1.11   Problems with the Current System

The current system consists of chess boards with pieces and paper if results want to be recorded. The main problem that was noticed was that as the club was only hosted on a single lunchtime a week, many games overran past lunchtime and so had to be stopped with no clear winner as the game had not finished. As the game was played on a physical board with pieces and the room that was used for the club is used for lessons as well, the current state of the game is not saved which resulted in the game effectively being cut short. This is incredibly frustrating, especially for higher-level players as their games tend to take longer to finish.

Another problem came from when in-school chess competitions occurred. These competitions required results to be recorded such that it could be determined who plays who in further rounds of the competition, i.e. what teams play against one another in the final. These are recorded at lunchtime on the board and then recorded later on paper. There are of course problems with such a system. Firstly, results can easily be lost as it is held in one location. This also means that only one person can view the results at the time, unless the results are published in an e-mail which takes time to produce. In addition, it does not store the final state of the game when finished which may be of interest to students and teachers alike.

A problem that has been noted by students is that even though they are willing to attend chess club, many are unable to attend at lunchtimes on a certain day because they have other commitments and so end up not attending at all despite their desire to do so. Many students would like to be able to play chess games outside of club hours amongst each other with results recorded at school.

1.12   Objectives

After the analysis of the current system and the requirements that would be needed for an improved system I have created SMART (Specific, Measurable, Achievable, Realistic and Timely) objectives which make clear to me - the developer what the client requires for their new system which has been determined to be feasible to complete and adheres to all of the letters of the SMART abbreviation.

The objectives of the new system are as follows:

1. The showing of an interactive chess board allowing movement of pieces by users which adheres to all chess rules.

    (a) When program loads, show default chess starting position (a new game).

    (b) Ensure only pieces of the moving player's corresponding colour are clickable.

    (c) When a piece is clicked calculate all of its legal moves.

        i. For each type of piece calculate where it could possibly go assuming there are no other pieces on the board (i.e. bishops diagonally, rooks horizontally/vertically etc.).

        ii. Taking into account where other pieces are on the board, calculate the possible moves for that piece.

        iii. If we are checking moves for a king, check that castling is possible. If we are checking moves for a pawn, check that en passant is possible.

        iv. Considering the moves that have been calculated, check that each position does not result in the moving player to be in check.

            A. If a move results in a player going into check then remove that move from the list of legal moves.

        v. When this calculation is done, cells in the table corresponding to the legal moves must be made clickable so that the player can move the appropriate piece there if they wish.

(d) After a piece is moved perform a check of the game state. If the state has changed (i.e. if a player has been put into check or the game has ended) then inform the user of this via a dialog.

(e) If the pawn has successfully reached the other side of the board, then open a dialog to let the user decide to what piece the pawn should be converted to.

2. Allow the editing of player's names via textboxes.

3. Allow a game that is currently in play to be saved.

(a) If there is no file path to save the game to, prompt the user for a path and filename to save a new JSON file to.

(b) If there is a file path and the JSON file is found, then save the game to the file.

(c) If there is a file path and the JSON isn't found, then prompt the user for a path and filename to save a new JSON file to.

(d) If a game has been successfully saved, convey this information to the user via a dialog.

(e) If the player names have not been filled in, then do not save the game and inform the user that they have to fill in the names in the form of a dialog.

(f) If the game has an ID then save it by replacing the element of the games array with the same name by searching for the appropriate ID with a binary search.

4. Allow a user to load a game and to continue to play it.

(a) If the JSON file is not found then prompt the user to input the path to the game.

(b) Display the list of games in a table to allow the user to know which game to choose.

    i. Information that must be shown in the table: ID, name of player 1 and 2, winner, moves made, last played.

ii. Using a quicksort algorithm, allow the user to sort the list of games based on parameters such as ID and the names of players.

Documented Design

2.1 Introduction

It has been decided that the new system will be a desktop application which can be used to play chess amongst two human players. It will be used by those who attend chess club to not only play games but to save, load and continue playing them. This program is to be developed in Python and for the graphical user interface Qt has been chosen, with the Pyside library serving as a python wrapper around the Qt GUI framework.

It will consist of a visual representation of a chess board which is held in a table GUI element. This will allow players to move pieces via clicking on a piece and then on the place it wants to move the piece to. It will contain several buttons, which all serve different functions. There will be functionality to start a new game and to save, and load, existing games. Saving a game will work by converting a python dictionary into JSON format and saving it in an existing JSON file by adding it to an array of dictionaries (games). If this file does not exist then the program will allow the user to select a location for a new JSON file to be created. This location will be saved as a setting so that the program can find this location again whenever it is relaunched.

Loading a game will open the JSON file from the location path specified by the user and scans the array of games, showing them as a table in a dialog so that the user can choose which game to open. If the location of the file does not exist, then the user will be given

the option of locating a different pre-existing JSON file to open from if it exists.

As already mentioned, the JSON file will hold an array of games. A game contains the data needed to load a game: pieces and their positions, total moves, state of the game and other variables that are used to determine whether certain moves are possible, such as castling. Each game will have a unique ID which will allow users to load games that haven't been finished and to carry them on. For the program's loading and saving feature to be most useful for the school's chess club, it is best to store the JSON file in the school's shared area such that all games can be loaded and saved from any computer that has this program installed.

2.2   Overall System Design

2.2.1   Input

The first type of input is selecting a piece on the board by clicking on a cell in the 8×8 table that contains a piece corresponding to the player's colour. At this point processing occurs and moves that are legal are added as cells that can be clicked on the table. After this, if a cell is clicked that is not a player's piece then the last clicked piece is moved to that location. If a cell is clicked that is a player's piece the process described above is repeated.

The second type of input is the player name's that must be set if the game is to be saved. Both names can be edited via 2 separate GUI TextEdit components.

The third type of input is to do with saving and loading games. A dialog asking for the location of a JSON file (or where to save it) allows the user to select the game file for the program. When the Load Button' game is pressed a dialog is shown where the user can choose a game to play from the file via a table.

2.2.2   Processing

The amount element of processing occurs when pieces are moved on the board. Once a user has used the GUI interface to declare what piece to move and to where it should be moved the main element of processing occurs. When this is done the program calculates all the possible legal moves for the opposing player's pieces. Then the GUI edits the table (which represents the chess board) such that the opponent can only select a cell in the table which represents a piece and its respective legal moves. In addition the program checks the game state and sees whether a player has won, or whether they are in check or if the game has ended in stalemate. If the game state has changed the user is informed through a dialog.

2.2.3   Storage

All data that is required to load and save games is stored in public attributes of the Board class. When a game is saved these attributes are collated into a dictionary which is added to an array of games as a JSON file. When a game is loaded a new Board class I created where all the attributes of the loaded game are put into the class from the JSON dictionary.

2.2.4   Output

In terms of output there are three categories of output that the user will see. The first is of course the current board. The board has a visual representation of all the pieces on the correct parts of the board which is put into an 8x8 table with all cells of equal size.

The second category is the state of the game. When the state of the game changes after a move, such as a player being in check or the game is over due to checkmate/stalemate, then the user is informed of this change by a dialog that appears. This dialog has a button to dismiss it to allow the players to continue to interact with the rest of the program,

and forces users to acknowledge the change in game state.

The third category is the load dialog. The load dialog greets the user with list of games as rows on a table. The columns of the table give information about each game that is in the JSON file that the program reads from, such as the game ID, players, last played, moved made, winner etc.

2.3   Heirarchy Charts

2.3.1   Top Level View
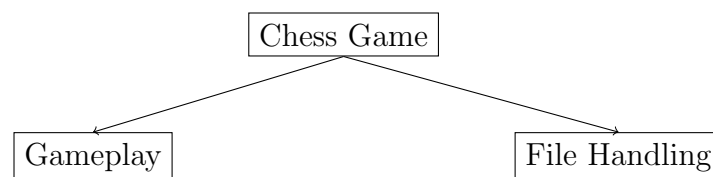
Figure 2.1: Top level hierarchy chart.

2.3.2   Second Level View
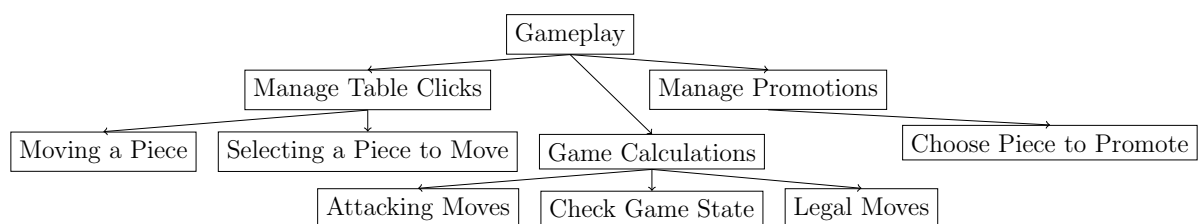
Figure 2.2: Second level hierarchy chart: Gameplay.
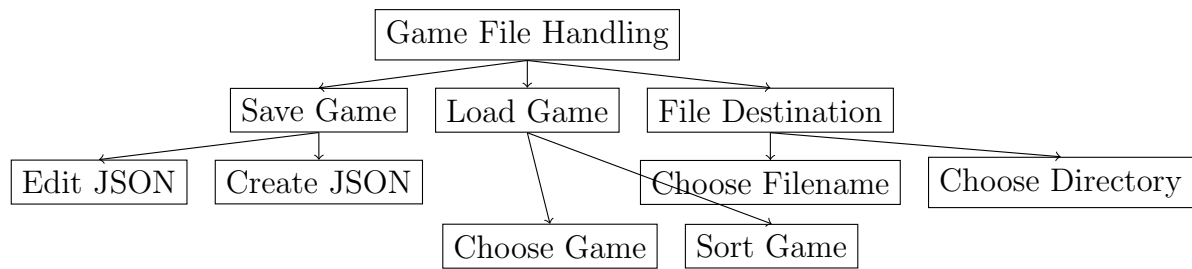
Figure 2.3: Second level hierarchy chart: Game File Handling.

2.4   Code Design

2.4.1   Class Diagrams
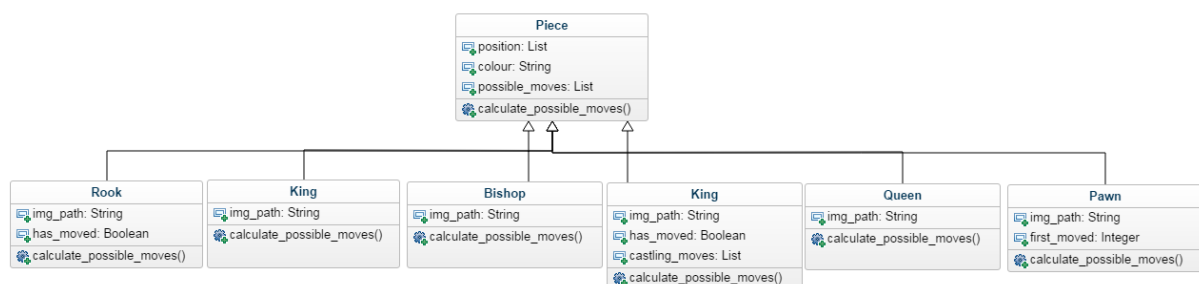
Figure 2.4: Class diagram for Piece and its subclasses.
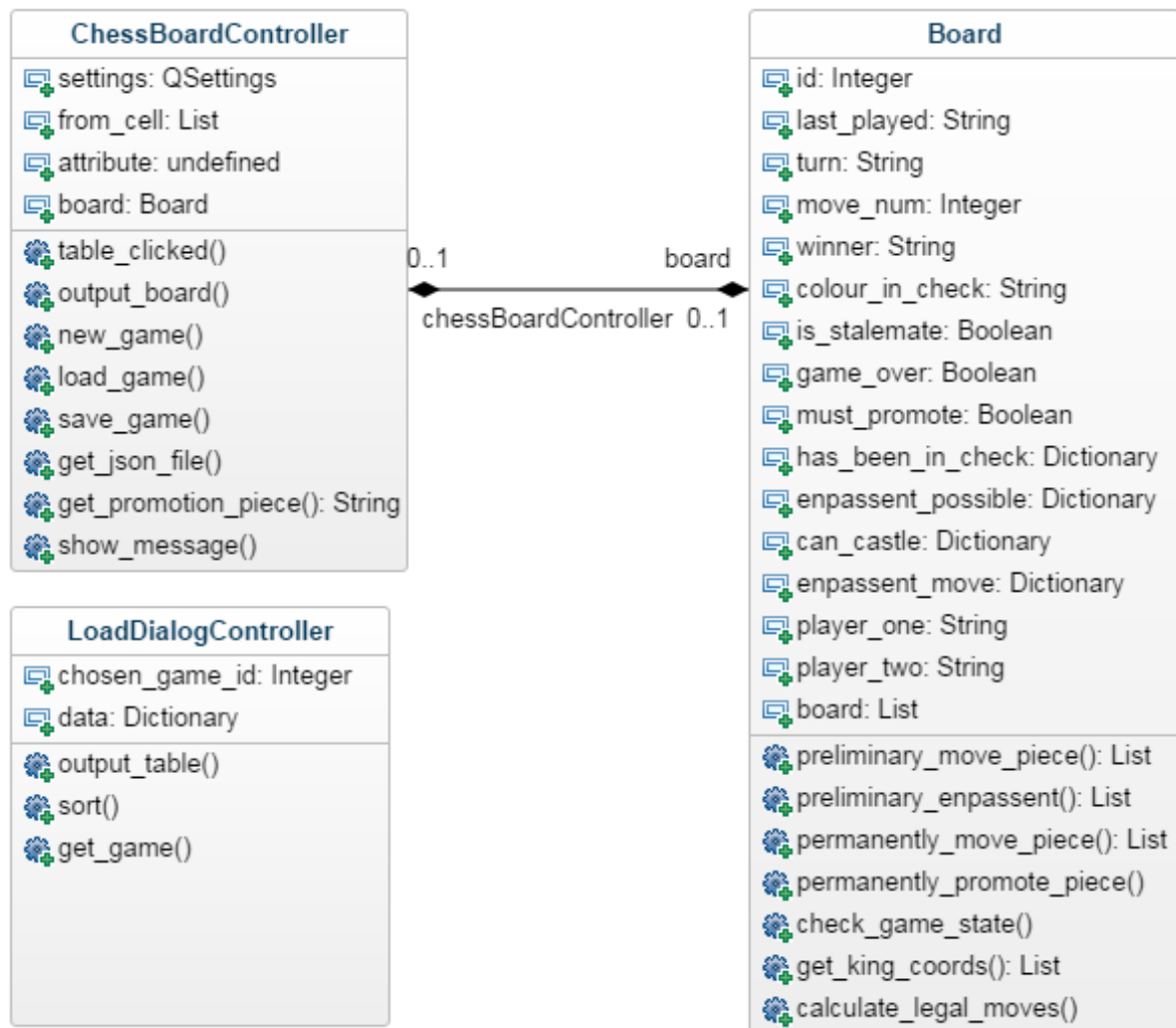
Figure 2.5: Class diagram for the controllers and model of the program.

2.4.2 File and Code Structure

As the program is in the form of a Graphical User Interface (GUI) I thought it would be appropriate to design my code around the Model-View-Controller (MVC) framework. The view of the MVC architecture is the definition of the GUI. The model is the logic of the game board and the controller is what accepts inputs and interactions with the view, which converts them to commands to the model. The results of the model processing this data is then given back to the view by the controller. The .ui file type is the file that the Qt framework generates after editing of a GUI occurs in the Qt Designer application. These are then converted to .py by a command line utility pyside-uic. I then collated

these files and put them into the views.py file.

```
/
└─ src
   ├─ ui .............................................. Contains all the UI files
   │  ├─ chessboard.ui
   │  ├─ loadgame.ui
   │  ├─ mainwindow.ui
   │  └─ promotion.ui
   ├─ algorithms.py ................................ Binary search and quicksort.
   ├─ board.py .................... Contains Board class: model for the controller.
   ├─ Chess.py ................................... Starts the program when run.
   ├─ controllers.py ................................ Contains all the controllers.
   ├─ pieces.py ...... Contains Piece class and its subclasses, used by Board class.
   ├─ views.py ....... Contains all the views: these are generated from the .ui files.
   ├─ resources_rc.py .................................... Contains image data.
   └─ requirements.txt ..... Contains list of all python packages that the program
      requires to run.
```

Figure 2.6: Code structure of the program.

2.4.3 Required Software

For this program to run via running the Chess.py file it is required that Python 3 up until 3.4 is installed on the computer that the program will be run from, anything higher is not supported. This is because the program relies on PySide being installed and it does not support versions newer than 3.4.

2.4.4 Code Documentation

In this section the comments to all the classes and functions in the program are shown. Views.py is not commented as it is simply a compilation of .ui files that have been compiled to Python classes and Chess.py is also not commented as it is simply boilerplate code to get the GUI to run. Similarly, resources_qrc.py just contains image data of the chess pieces.

26

algorithms.py

```python
def quick_sort(array, low, high):
    """Sorts a list recursively.

    Args:
        array (list): the list to be sorted.
        low (int): index of lowest item to be sorted.
        high (int): index of highest item to be sorted.
    """
    def partition(array, low, high):
    """Partitions array using a pivot value.

    Args:
        array (list): the list to be sorted.
        low (int): index of lowest item to be sorted.
        high (int): index of highest item to be sorted.

    Returns:
        The value of the pivot.
    """


def binary_search(search_term, array):
    """Searches for an item in an already sorted list.

    Args:
        search_term (str): term to be searched for in the list.
        array (list): list to be searched.

    Returns:
        True if found, False otherwise.
    """
```

board.py

```python
class Board(object):
    """Class which manages the pieces on the board.

    This is the model for the ChessBoardController class.
    The primary function of this class is to allow the controller to determine
    the legal moves a given piece can make and to move pieces on the board, whilst
    performing checks of the game state after every move.

    Attributes:
        id (int): the unique identifier for a particular game.
        last_played (str): The date at which the game was last saved.
        turn (str): The colour of the current player's turn.
        move_num (int): how many moves have been made in the game.
        winner (str): The name of the winner.
        colour_in_check (str): if a player is in check, their colour is held in this
    ↪    variable.
        is_stalemate (bool): shows if game is in stalemate or not.
        game_over (bool): shows if game is over or not.
```

```python
        must_promote (bool): true if a player must promote their pawn, false
↪   otherwise.
        enpassant_possible (dict): shows if black and white can complete an en
↪   passant move.
        enpassant_move (dict): stores the en passant move for both colours if they
↪   exist.
        player_one (str): stores the name of the first player (white).
        player_two (str): stores the name of the second player (black).
        board (list): an 8*8 2D list which maps to pieces on the board.
    """

    def __init__(self, game=None):
        """Loads a game if given one, otherwise intialises a new game.

        Args:
            game (dict): contains all data needed to load a game into a Board object.
        """

    def new_board(self):
        """Creates a new board."""

    def preliminary_move_piece(self, chess_board, old_coords, new_coords):
        """"Will move a piece temporarily. e.g. used to see if piece puts itself in
        ↪   check.

        Args:
            chess_board (list): the board to be used to move the piece.
            old_coords (list): coordinates of the piece to be moved.
            new_coords (list): coordinates of where the piece will be moved to.

        Returns:
            list: returns a chess board with the piece moved.
        """

    def preliminary_enpassant(self, chess_board, old_coords, new_coords,
    ↪   removed_coords):
        """"Will perform enpassant temporarily e.g. used to see if piece puts itself
        ↪   in check.

        Args:
            chess_board (list): the board to be used to move the piece.
            old_coords (list): coordinates of the piece to be moved.
            new_coords (list): coordinates of where the piece will be moved to.
            removed_coords (list): coordinates of the piece to be taken.

        Returns:
            list: returns a chess board with the piece moved.
        """

    def permanently_move_piece(self, chess_board, old_coords, new_coords):
        """
        Moves a piece on the board permanently.
        It will also check for change in game state (checkmate, check, stalemate).

        Args:
            chess_board (list): the board to be used to move the piece.
            old_coords (list): coordinates of the piece to be moved.
            new_coords (list): coordinates of where the piece will be moved to.
```

```python
    Returns:
        list: returns a chess board with the piece moved.
    """

def permanently_promote_piece(self, type, coords):
    """Promotes a pawn to a piece of a certain type.

    Args:
        type (str): Name of the type of piece to promote to.
        coords (list): coordinates of the pawn to be promoted.
    """

def check_game_state(self):
    """Completes a check of the state of the game."""

def get_king_coords(self, colour, board):
    """Finds the coords of the king depending on its colour.

    Args:
        colour (str): color of the king to find.
        board (list): the board to find the kind from.

    Returns:
        list: coordinates of the king in the form [x,y] (0-based).
    """

def calculate_legal_moves(self, piece):
    """Gets all the legal moves of a piece and returns them.

    Args:
        piece (Piece): The piece to calculate the legal moves for.

    Returns:
        list: A list of all the legal moves a piece can make.
    """

def get_attacking_moves(self, piece, board):
    """
    Get all the attacking moves of a piece and return them.
    The difference between this function and get_legal_moves() is that
    this shows you what pieces can force a check while the other functions
    tells you whether the piece can move there.

    Args:
        piece (Piece): The piece to calculate the legal moves for.
        board (board): board used to determine attacking moves.

    Returns:
        list: A list of all attacking moves a piece can make.
    """

def calculate_is_checkmate(self, colour, board):
    """Finds out if a player is in checkmate.

    Args:
        colour (str): Colour that we are checking for if they are in checkmate.
        board (list): the board of the game being played.
```

```python
            Returns:
                bool: True if the game is in a state of checkmate, False otherwise.
            """

    def calculate_is_stalemate(self, board):
        """Finds out if a game is in stalemate.

        Args:
            board (list): the board of the game being played.

        Returns:
            bool: True if the game is in a state of stalemate, False otherwise.
        """

    def is_in_check(self, colour, possible_board):
        """Checks if the king of the corresponding colour is in check.

        Args:
            colour (str): colour that we are checking if they are in check.
            possible_board (list): the board of the game being played.

        Returns:
            bool: True if the player is in check, False otherwise.
        """

    def get_castling_moves(self, king, original_board):
        """Finds castling moves, if they exist, for a given king.

        Args:
            king (King): the king that we are finding castling moves for.
            original_board (list): the board of the game being played.
        """

    def can_enpassant(self, pawn, possible_board):
        """Checks if en passant is possible, and assigns the en passant move if so.

        Args:
            pawn (Pawn): the pawn that we are determining if it can perform en
↪    passant.
            possible_board (list): the board of the game being played.

        Raises:
            IndexError: raised when index is -1 or 8 as these indices do not exist in
↪    possible_board.
            """
```

controllers.py

```python
class MainWindowController(QtGui.QMainWindow, views.MainWindow):
    """Controller for the main window of the application"""


class ChessBoardController(QtGui.QWidget, views.ChessBoard):
    """Controller for the chess board.
```

```
    Attributes pertaining to the views.ChessBoard class are not listed here as there
↪   are too many
    and they merely refer to Qt GUI Objects.

    Attributes:
        board (Board): instance of the Board class which serves as the model for the
↪   controller.
        settings (QSettings): a class used to store settings (namely file path) on
↪   the host computer.
        from_cell (list): stores the coordinates of the cell that was chosen to move
↪   from.
    """

    def __init__(self):
        """Initialises necessary variables, connects click events to methods and
        ↪   presents a new game to the user."""

    def table_clicked(self):
        """Handler for when table is clicked.

        When a piece has been selected to move then its legal moves are calculated.
        These moves are then made clickable and highlighted yellow.
        If a user selects any of the highlighted colours then the piece is
↪   transferred to that position
        and a check of the game state occurs where any changes are shown in a msg
↪   box.
        Promotions, if necessary, are called in this method.
        """

    def output_board(self, legal_moves=[]):
        """Output the board onto the GUI

        Uses the Board.board (2D list) to help populate the QTableWidget table.
        All pieces of the moving player are made clickable on the table.
        In addition, if there are legal moves specified they are all made clickable.

        Args:
            legal_moves (list): list of the legal moves a piece can make.
        """

    def new_game(self):
        """Creates a new game with initial board setup."""

    def load_game(self):
        """Loads a game from a JSON file. If there is no JSON file the user is
        ↪   prompted for one.

        Raises:
            IOError: raised when there is an error loading a file.
            FileNotFoundError: raised when a file is not found.
            TypeError: raised when there is an error loading a file.
            KeyError: raised when there is corruption in the JSON file.
        """

    def save_game(self):
        """Saves a currently played game, either in an existing JSON file or a new
        ↪   one.
```

```
        Raises:
            IOError: raised when there is an error loading a file.
            FileNotFoundError: raised when a file is not found.
        """

    def get_load_path(self):
        """Gets the JSON file to load"""

    def get_save_path(self):
        """Gets the desired path and filename of the file to save."""

     def get_promotion_piece(self):
        """Gets the piece that needs to be promoted"""

     def show_message(self, msg):
        """If there are errors, they are shown in a message box.

        Args:
            msg (str): message for the user to be shown.
        """


class LoadDialogController(QtGui.QDialog, views.LoadDialog):
    """Controller for the load dialog

    When 'Load Game' is pressed the user is shown this dialog if a JSON file is found
↪    and has been loaded.
    The user is shown a table where they can sort the games and can also choose which
↪    game to load.

    Attributes:
        chosen_game_id (int): id of the game that was chosen
        data (dict): List of games.
    """

     def __init__(self, data):
        """Necessary variables are initialised and table is shown to the user."""

     def output_table(self):
        """List of games is shown to the user with this method."""

     def sort(self):
        """Sorts the list of games"""

     def get_game(self):
        """When game is chosen the ID is then found."""
```

pieces.py

```
class Piece(object):
    """Base class for all chess pieces

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): Colour of the piece, either "White" or "Black"
```

```python
    """

    def __init__(self, position, colour):
        """This constructor initialises the class variables and also calculates all
        ↪   possible moves for the piece."""

        @position.setter
        def position(self, value):
            """ This setter calculates the new possible moves once the position has
            ↪   changed.

            Args:
                value (list): coordinates on the board in the form [x, y]
            """

    def calculate_possible_moves(self):
        """Shows us the coords it can go to assuming that there are no other pieces
        ↪   on the board

        Raises:
            NotImplementedError: Method not overridden in subclass.
        """


class Rook(Piece):
    """Class for a Rook

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): colour of the piece, either "White" or "Black"
        img_path (str): path to the image of the piece
        has_moved (bool): Denotes whether the piece has moved before.
    """

    def __init__(self, position, colour, has_moved=False):
    """
    This constructor initialises the class variables and also calculates all possible
↪   moves for the piece.
    In addition the image path is added.
    """

        def calculate_possible_moves(self):
            """Calculates all the possible moves for a rook in a certain position."""


class Knight(Piece):
    """Class for a Knight

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): colour of the piece, either "White" or "Black"
        img_path (str): path to the image of the piece
    """

    def __init__(self, position, colour):
        """
        This constructor initialises the class variables and also calculates all
↪   possible moves for the piece.
```

33

```python
        In addition the image path is added as an attribute self.img_path.
        """

    def calculate_possible_moves(self):
        """Calculates all the possible moves for a knight in a certain position."""


class Bishop(Piece):
    """Class for a Bishop

    Attributes:
        position (list): coordinates on the board in the form [x, y]
    colour (str): colour of the piece, either "White" or "Black"
    img_path (str): path to the image of the piece
    """

    def __init__(self, position, colour):
    """
    This constructor initialises the class variables and also calculates all possible
↪   moves for the piece.
    In addition the image path is added as an attribute self.img_path.
    """

    def calculate_possible_moves(self):
        """Calculates all the possible moves for a bishop in a certain position."""


class Queen(Piece):
    """Class for a Queen

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): colour of the piece, either "White" or "Black"
        img_path (str): path to the image of the piece
    """

    def __init__(self, position, colour):
        """
        This constructor initialises the class variables and also calculates all
↪   possible moves for the piece.
        In addition the image path is added as an attribute self.img_path.
        """

    def calculate_possible_moves(self):
        """Calculates all the possible moves for a queen in a certain position."""


class King(Piece):
    """Class for a King

    Attributes:
        position (list): coordinates on the board in the form [x, y].
        colour (str): colour of the piece, either "White" or "Black".
        img_path (str): path to the image of the piece.
        has_moved (bool): denotes whether piece has moved or not.
        castling_moves (list): list of all castling moves possible.
    """
```

```python
    def __init__(self, position, colour, has_moved=False, castling_moves=[]):
        """
        This constructor initialises the class variables and also calculates all
↪   possible moves for the piece.
        In addition the image path is added as an attribute self.img_path.
        """

     def calculate_possible_moves(self):
        """Calculates all the possible moves for a king in a certain position."""


class Pawn(Piece):
    """Class for a Pawn

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): colour of the piece, either "White" or "Black"
        img_path (str): path to the image of the piece
        first_moved (int): denotes at what stage (move) in the game the the piece was
↪   first moved.
    """

    def __init__(self, position, colour, first_moved=0):
        """
        This constructor initialises the class variables and also calculates all
↪   possible moves for the piece.
        In addition the image path is added as an attribute self.img_path.
        """

     def calculate_possible_moves(self):
        """Calculates all the possible moves for a pawn in a certain position."""
```

2.5 Input Validation

Field	Validation	Description	Error Message
Player names	Presence	Check for their presence	"Please fill in the player names"
Path to JSON file	Valid	Checks that file exists at specified location	"File not found"
Name of new JSON file	Presence/Valid	Must not contain special characters	"File not saved"
Game to load	Presence	Make sure ID corresponds to existing game	Game not loaded
Move to make	Move legality	Checks if a proposed move is valid	Illegal moves are not marked yellow and are not clickable on the table

Table 2.1: Table of information on input validation.

2.6 Algorithm Commentary

In this section, algorithms used in the code base that are considered complex are fully explained.

2.6.1   Binary Search

```python
def binary_search(search_term, array):
    """Searches for an item in an already sorted list.

    Args:
        search_term (str): term to be searched for in the list.
        array (list): list to be searched.

    Returns:
        True if found, False otherwise.
    """
    half_array = int(len(array)/2)
    if search_term == array[half_array]:
        return True
    elif len(array) == 1:
        return False
    elif search_term > array[half_array]:
        return binary_search(search_term, array[half_array:])
    else:
        return binary_search(search_term, array[:half_array])
```

Figure 2.7: Source code for the recursive binary search algorithm.

As can be noted, this algorithm is a recursive function, with recursive calls at lines 17 and 19. The function is given a sorted array and compares the middle value of the array to the search term. The base case is met when the search term is found (line 13) or if the length of the array to be searched is 1 **and** the search term is not found (line 15). If that is not the case then if the search term is larger than the middle value, then the left half of the array is passed into the same function. Otherwise the right half is passed into the same function. This carries on until the base case is met. When it is the value of `True` or `False` is passed up the call stack such that the first call of the function will return `True` or `False`.

2.6.2   Quicksort

Below is the code for the recursive quicksort algorithm:

```python
1    def quick_sort(array, low, high):
2        """Sorts a list recursively.
3
4        Args:
5            array (list): the list to be sorted.
6            low (int): index of lowest item to be sorted.
7            high (int): index of highest item to be sorted.
8        """
9        def partition(array, low, high):
10           """Partitions array using a pivot value.
11
12           Args:
13               array (list): the list to be sorted.
14               low (int): index of lowest item to be sorted.
15               high (int): index of highest item to be sorted.
16
17           Returns:
18               The value of the pivot.
19           """
20           i = low + 1
21           pivot = array[low]
22           for j in range(low+1, high+1):
23               if array[j] < pivot:
24                   array[j], array[i] = array[i], array[j]
25                   i += 1
26               array[low], array[i-1] = array[i-1], array[low]
27               return i - 1
28
29       if low < high:
30           pivot = partition(array, low, high)
31           quick_sort(array, low, pivot-1)
32           quick_sort(array, pivot+1, high)
```

Figure 2.8: Source code for the recursive quicksort algorithm.

It can be noted that there is an inner function defined at line 9. Instead of defining it as another function I decided to use it as an inner function to avoid code repetition and because it has no use outside of the quicksort function. The function accepts an unsorted list, and the index of the lowest and highest element of the list to sort. The partition function is then called, where the pivot value is the first element in the list. The array is changed such that all elements smaller than the pivot are put before it, and all elements bigger are put after it. Then the quicksort function is called twice. Once for values below the pivot and for values after. Once the base case has been met for each recursive call, the list is sorted. It does not need to be returned as an array (list) in python is a mutable change and is thus changed in its scope.

38

2.7   User Interface
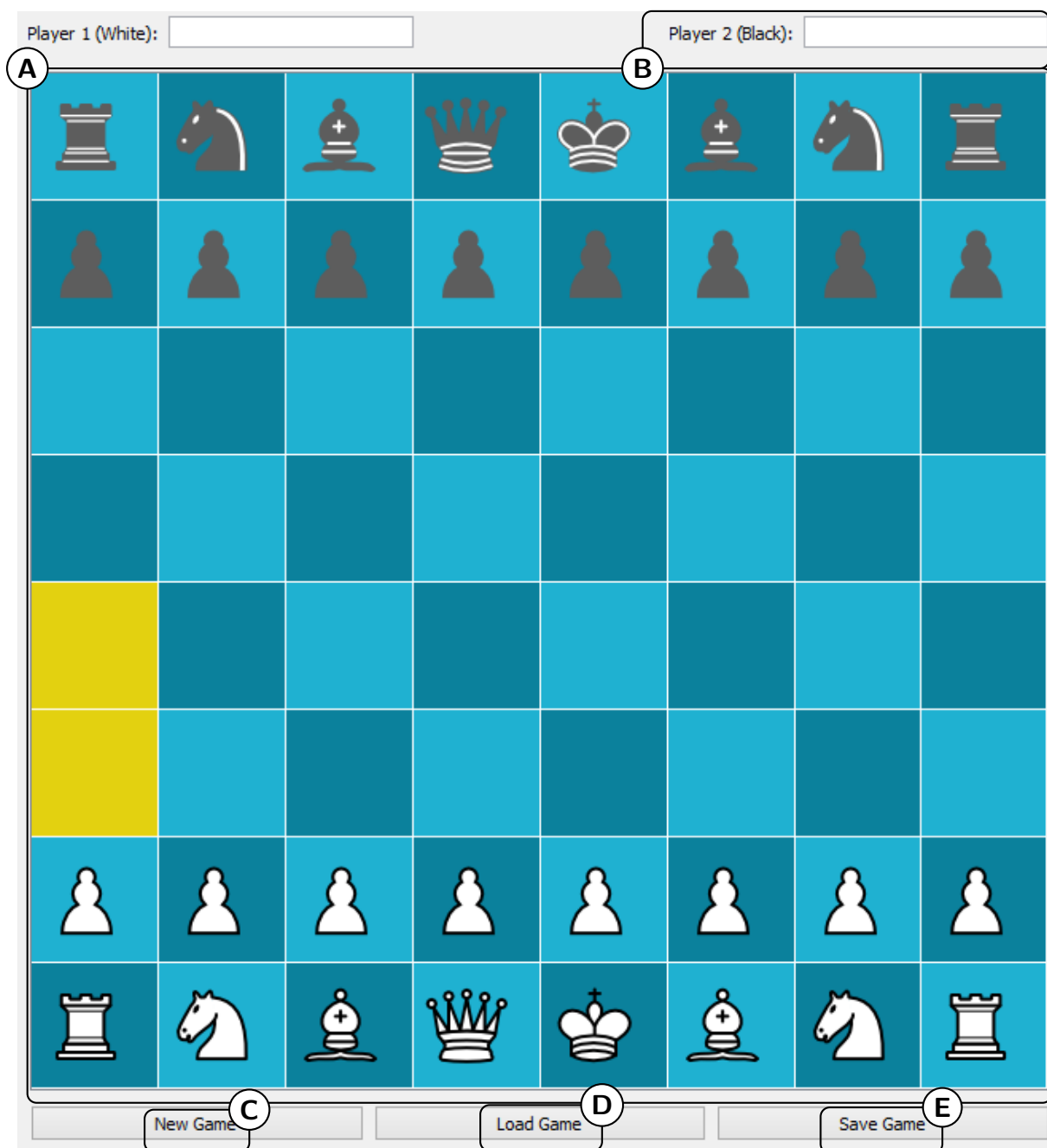
Figure 2.9: UI of the main program.

(A) This is an 8×8 table where input is carried out by clicking a piece then clicking again to a place where the piece is desired to be moved to. Only pieces for the player that is currently moving can be clicked. When it is clicked, all of its legal

moves are made yellow, and its cells are made clickable. All the cells of illegal moves are made unclickable, which simplifies validations of moves proposed a user.

(B) This is where the name of the second player (black) is added. To the left is where the name of the first player can be added.

(C) Pressing this button starts a new game, with the board the same as in figure 1.1.

(D) Pressing this button results in a dialog being shown, where the user can choose which game to load. If the game file is not found then the user will be prompted to insert the file's path.

(E) Pressing this button saves the game or prompts the user for a path to save the file in.
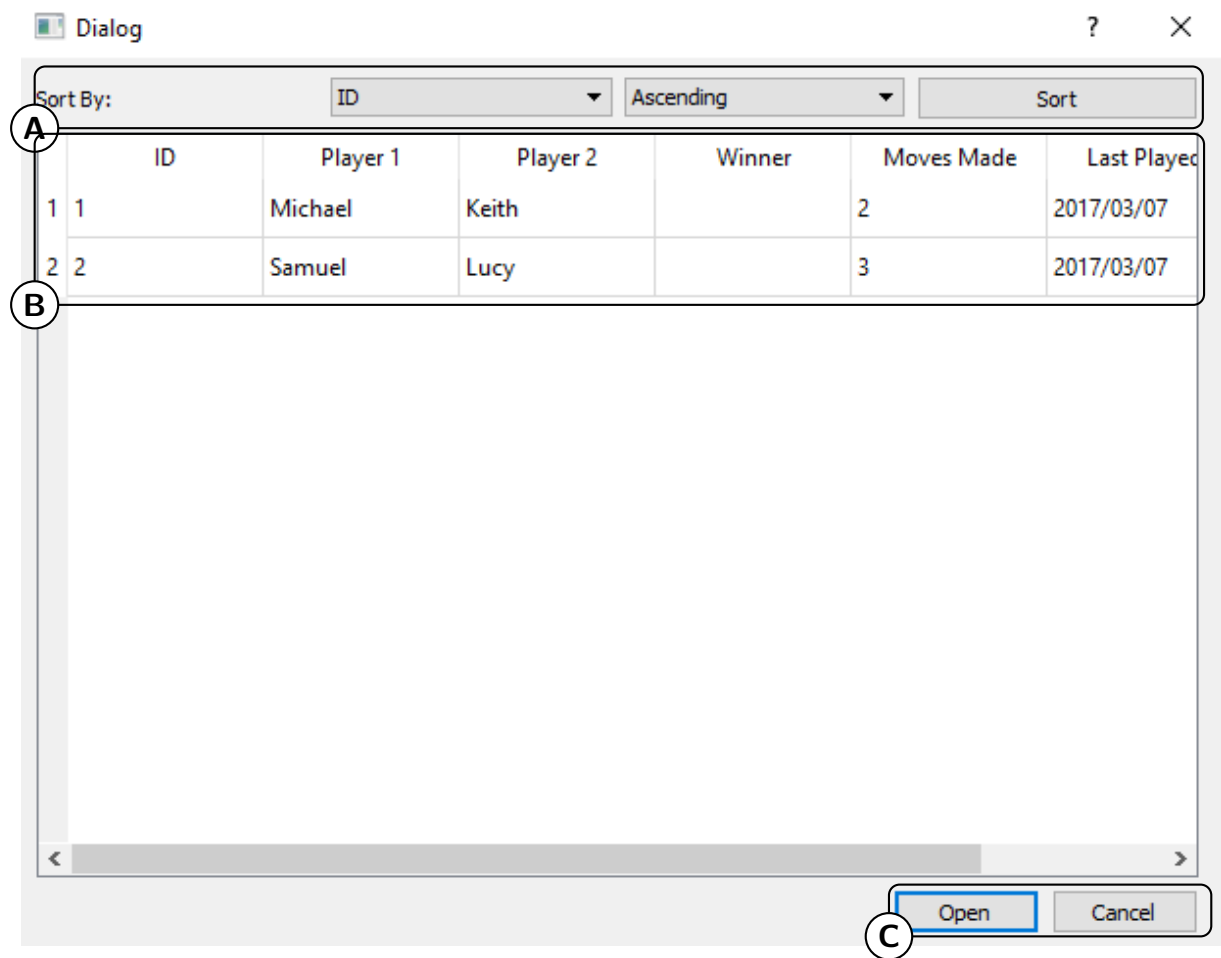
Figure 2.10: The load dialog of the program.

(A) This section comprises of two dropdown lists, one of which determines what to sort the list of games by and the second determines whether they should be shown in ascending or descending order.

(B) This table shows the list of all of the games that are saved in the JSON file.

(C) The user can either cancel or open a game. For a game to be successfully loaded, the user must click on a game in the list and then press the 'Open' button.
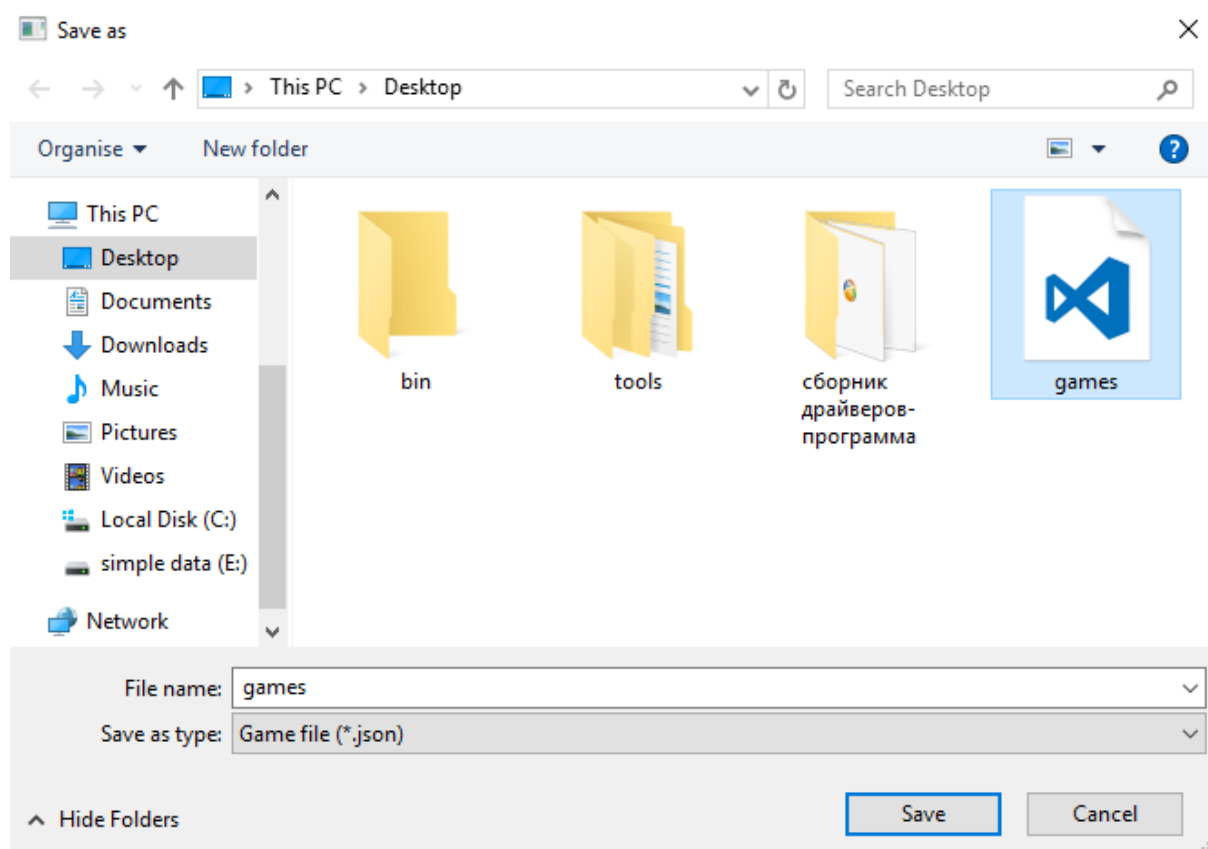

Figure 2.11: A "Save As" dialog used to save a new game file.
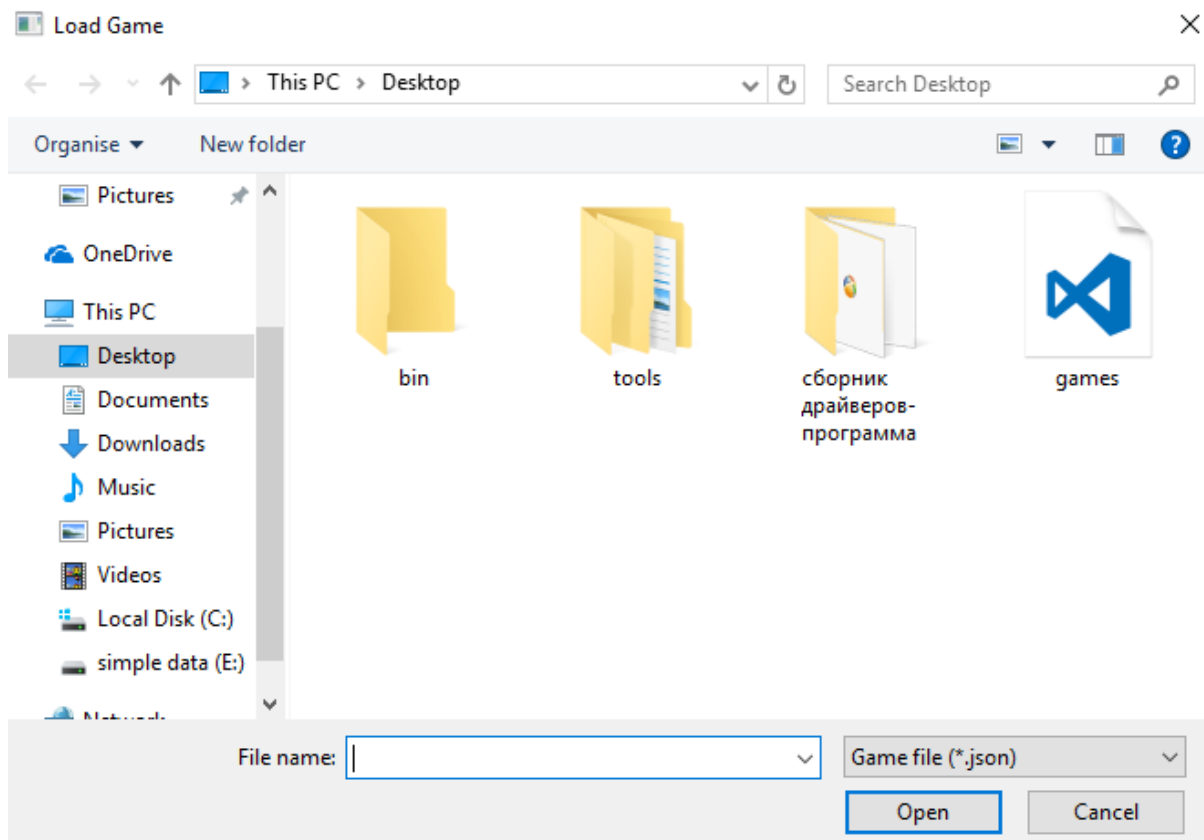
Figure 2.12: A file chooser dialog used to choose the game file to load.

Testing

3.1   Introduction

To be able to evaluate the effectiveness of the solution in its fulfilment of the objectives outlined in section 1.12 on page 17, it is necessary to conduct testing of the new system. Thus a test plan has been created, which when carried out, will determine whether the solution has met the requirements that have been agreed by the client and developer. The evidence (screen shots) of these tests have been included in the Appendix and start on page 49.

3.2   Testing Plan

ID	Obj ID	Test Data	Description	Expected Result	Evidence	Passed
1	1a	Start Program.	Check that program is usable.	A new game	Figure **??**	Yes
2	1b	Click piece.	Check that other person's piece can't be moved	Not clickable.	Figure A.1	Yes
3	1c	Click queen.	Checks that the queen moves as expected	Legal moves shown in yellow.	Figure A.2	Yes

4	1c	Click king.	Checks that the king moves as expected	Legal moves shown in yellow.	Figure A.3	Yes
5	1c	Click bishop.	Checks that the bishop moves as expected	Legal moves shown in yellow.	Figure A.4	Yes
6	1c	Click knight.	Checks that the knight moves as expected	Legal moves shown in yellow.	Figure A.5	Yes
7	1c	Click rook.	Checks that the rook moves as expected	Legal moves shown in yellow.	Figure A.6	Yes
8	1c	Click pawn.	Checks that the pawn moves as expected	Legal moves shown in yellow.	Figure A.7	Yes
9	1c	Click pawn.	Checks that the pawn can do en passant	En passant move shown in yellow.	Figure A.8	Yes
10	1c	Click king.	Checks that king can castle	Castling move shown in yellow	Figure A.9	Yes
11	1d	Put player into check.	Correctly identifies change in game state	Dialog shown when game state changes.	Figure A.10	Yes
12	1d	Put player into checkmate.	Correctly identifies change in game state	Dialog shown when game state changes.	Figure A.11	Yes
13	1d	Put game into stalemate.	Correctly identifies change in game state	Dialog shown when game state changes.	Figure A.12	Yes
14	1e	Move pawn.	Checks that promotion feature works.	Dialog allowing user to choose a new piece.	Figure A.13	Yes
15	2	"Keith", "Michael"	Allows names of players to be added	Names correctly saved.	Figure A.14	Yes

16	2	"Simon", "Alice"	Allows names of players to be changed	Changed names correctly saved.	Figure A.15	Yes
17	3a	Press "Save Game" when there is no file path.	Allow a game that is currently in play to be saved	Dialog allowing file path to be chosen.	Figure A.16	Yes
18	3b/d	Press "Save Game" when there is a file path.	Allow a game that is currently in play to be saved	Dialog stating that game has been saved.	Figure A.17	Yes
19	3c	Press "Save Game" when there is a file path but no file there.	Allow a game that is currently in play to be saved	Dialog allowing file path to be chosen.	Figure A.18	Yes
20	3e	Press "Save Game" when player names are not filled.	Save only when required data is filled.	Dialog requesting that names be filled.	Figure A.19	Yes
21	3f	Open existing game and make a move.	Check that existing games can be overwritten.	Game saved	Figure A.20	Yes
22	4a	Press "Load Game" when file does not exist.	Allow user to select the game file to load from.	Dialog requesting for path to file	Figure A.21	Yes
23	4bi	Press "Load Game"	Check that list of games is shown	Load dialog present.	Figure A.22	Yes
24	4bii	Choose parameters and press sort	Check that games can be sorted.	Correctly sorted list of games.	Figure A.23	Yes

Table 3.1: List of tests that been conducted and their results.

Evaluation

4.1 Introduction

In this section, the user is asked for feedback. In particular, the user is asked to determine whether the new system has met the SMART objectives that were agreed upon in section 1.12. Furthermore, the user explains what improvements could have been made to the new system.

4.2 User Feedback

4.2.1 User Acceptance Testing

The UAT involves the user testing the core functionality of the system to ensure it meets their requirements as specified in the SMART objectives. It is assumed that testing has already taken place and there are no major bugs. References to figures are made where it is deemed that it helps illustrate the point the user is trying to make.

The showing of an interactive chess board allowing movement of pieces by users which adheres to all chess rules. (Objective 1)

When the game first runs the default setting is a new game, the board is displayed ready

to play with spaces for both players to enter names. The interface is very clear and intuitive. Selecting a piece results in all its legal moves being displayed in yellow in their corresponding cells. Legal moves are calculated correctly with no bugs noticed in this regard. When the game state changes, such as a player being put into check, this change is shown quickly to me via the use of a dialog. Special types of moves such as en passant, castling and promotion are implemented quickly and intuitive to perform.

Editing of player names (Objective 2)

Editing of the names of the two players is very simple to do and the textboxes to change names are clearly shown at the top of the program. When the game is saved changes in names are correctly saved.

Saving a game (Objective 3)

When saving the game I was afforded the opportunity to select the filename and path for the new game file. Once this was done I was informed that the game was successfully saved. As I continued playing I periodically saved the game, which was seamless and each time I was informed of its success. In addition, I could create completely new games and save them in the same game file.

Loading a game (Objective 4)

Loading games that I had previously played was a simple procedure. I was shown a list of all the games that were saved and could sort them in ascending/descending order based on a wide range of parameters. Once I chose a game it loaded quickly and could continue playing it. Saving changes worked correctly.

Overall evaluation

This project definitely meets all the objectives that were agreed to at the start of the project. Thus, I have no suggestions in terms of features that could be added to meet the current requirements. However, there are some things outside of the scope of the requirements that would be nice to have. Due to the ubiquity of tablets and smartphones,

it would be nice to have an application that would work on those form factors, not just on a laptop or personal computer. In addition, it would be nice if there was a server for the storage of games, so that students could play outside of the school system in their own time.

4.2.2   Possible Improvements

The user identified that he would like to see this new system to be ported to handheld devices and for the storage of games to be held on a server, to allow games to be played anywhere at any time. To do this a server could be setup with a Linux distribution (e.g. Ubuntu) and a web server such as NGINX, where Python could be used for the back-end code which connects to a MySQL database. A REST API could be created such that it can be easy to make a front-end interface for all types of devices. Applications for Android and Windows/Linux could be written in Java, whilst for iOS, Swift or Objective-C would be used for development. Obviously doing all this could take a lot of time to learn these languages and their associated frameworks for development. There is a simpler method of creating a REST API and then connecting it to a Single Page Application (SPA) using a JavaScript framework such as Angular. This would allow the application to be used by any device that has access to a browser.

Appendix

A.1 Testing Evidence
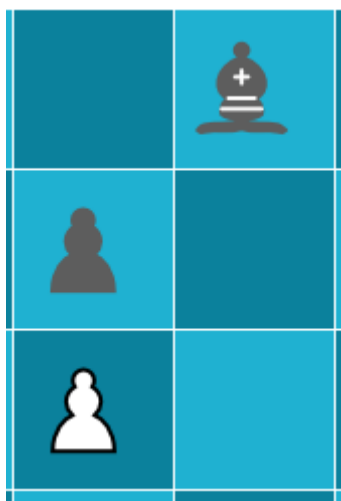
Figure A.1: Test 2 - white to move, black pieces are greyed out (not clickable).
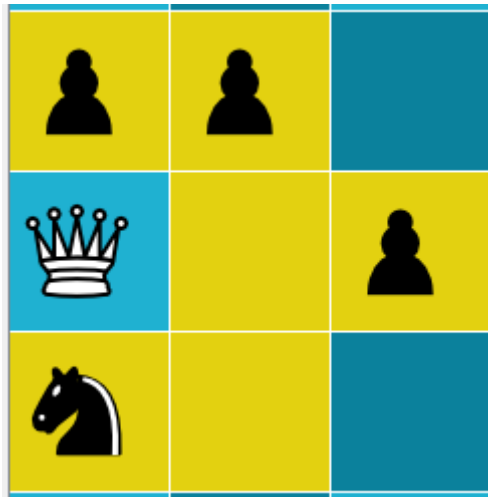
Figure A.2: Test 3 - legal moves of a queen.
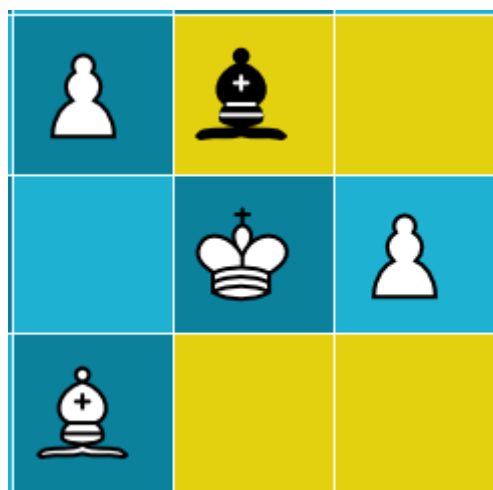
Figure A.3: Test 4 - legal moves of a king.

Figure A.4: Test 5 - legal moves of a bishop.
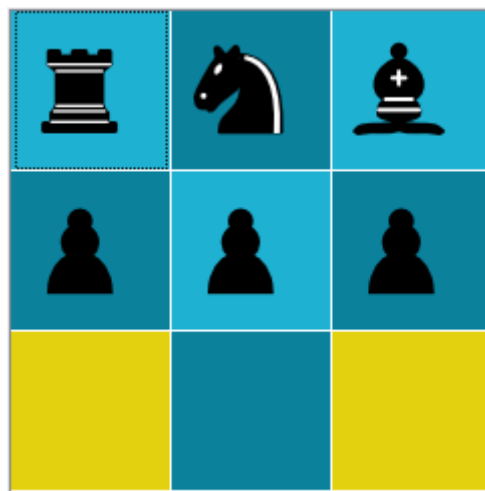
Figure A.5: Test 6 - legal moves of a knight.

Figure A.6: Test 7 - legal moves of a rook.
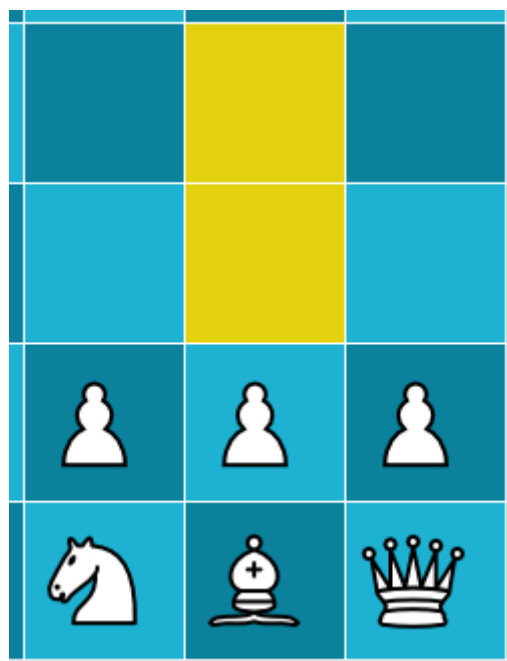
Figure A.7: Test 8 - legal moves of a pawn.
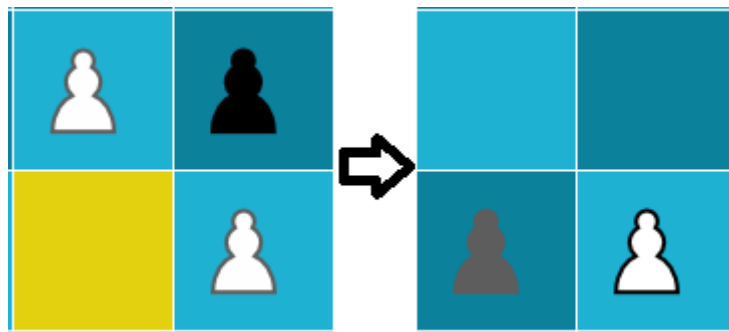
Figure A.8: Test 9 - en passant move.
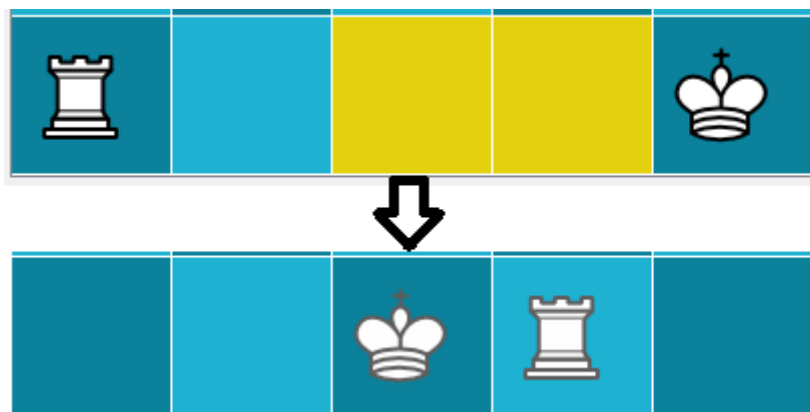
Figure A.9: Test 10 - castling move.
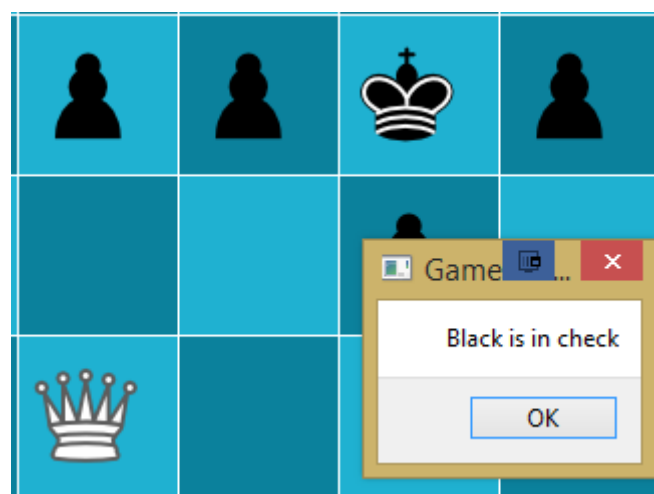
Figure A.10: Test 11 - dialog notifying user of check.

Figure A.11: Test 12 - dialog notifying user of checkmate.

Figure A.12: Test 13 - dialog notifying user of stalemate.
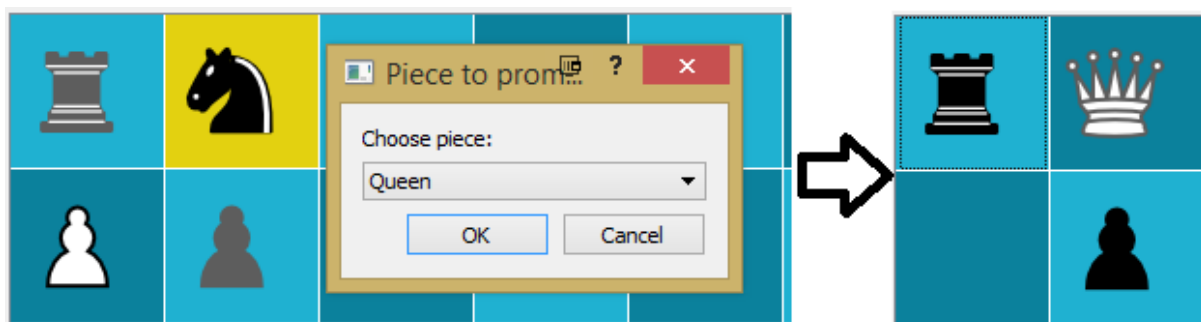
Figure A.13: Test 14 - check that promotion works.
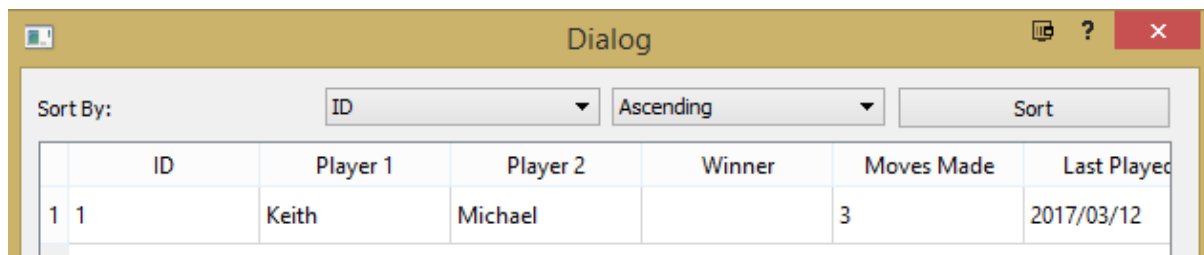
Figure A.14: Test 15 - check that game saved only when required data is filled.
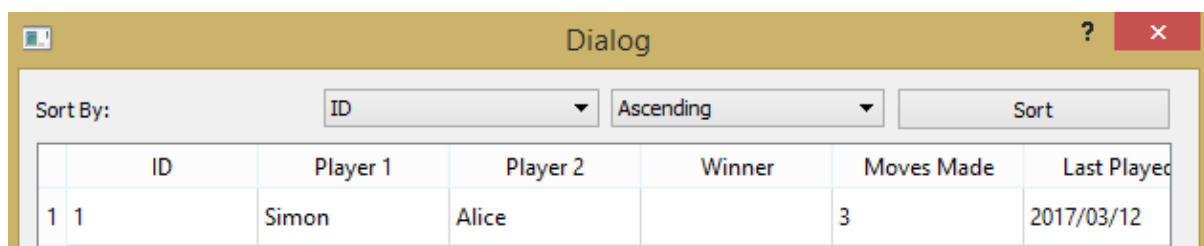
Figure A.15: Test 16 - check that game saved only when required data is filled.
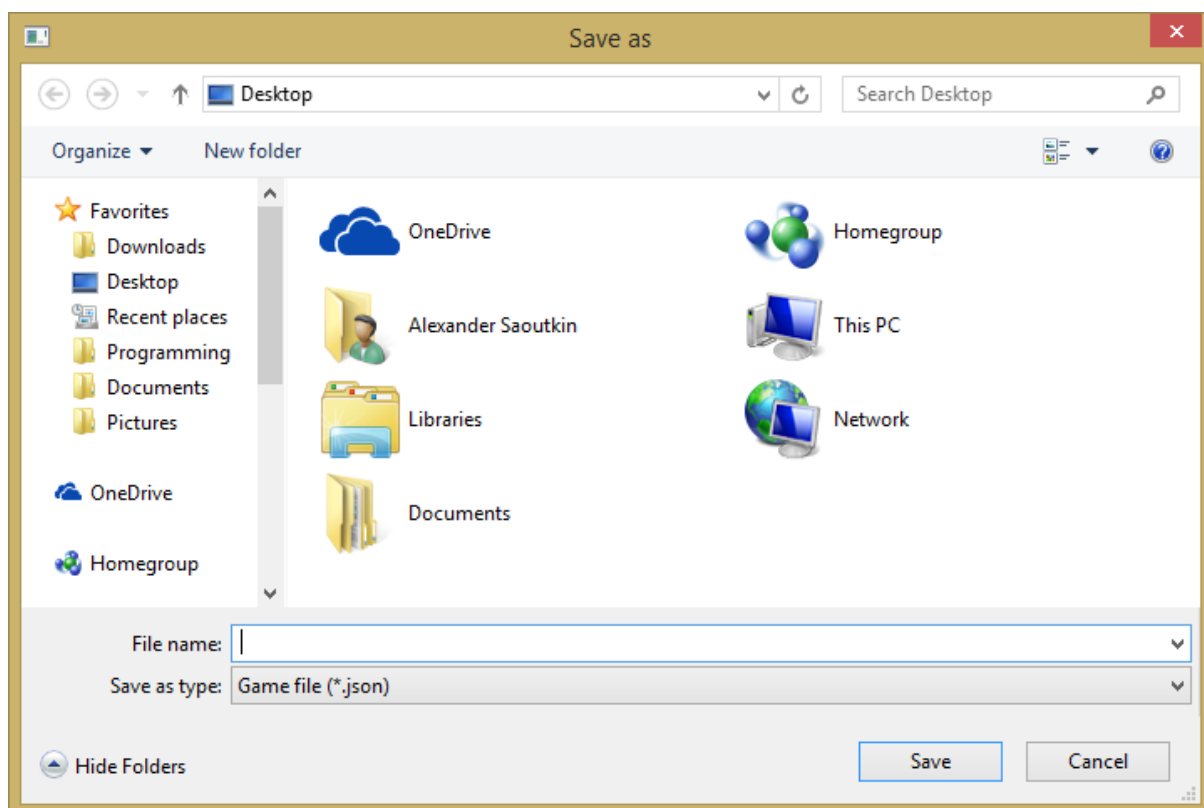
Figure A.16: Test 17 - "Save as" dialog to save game.
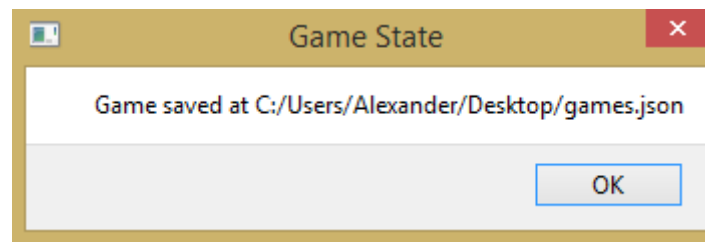
Figure A.17: Test 18 - dialog showing that game has been saved.
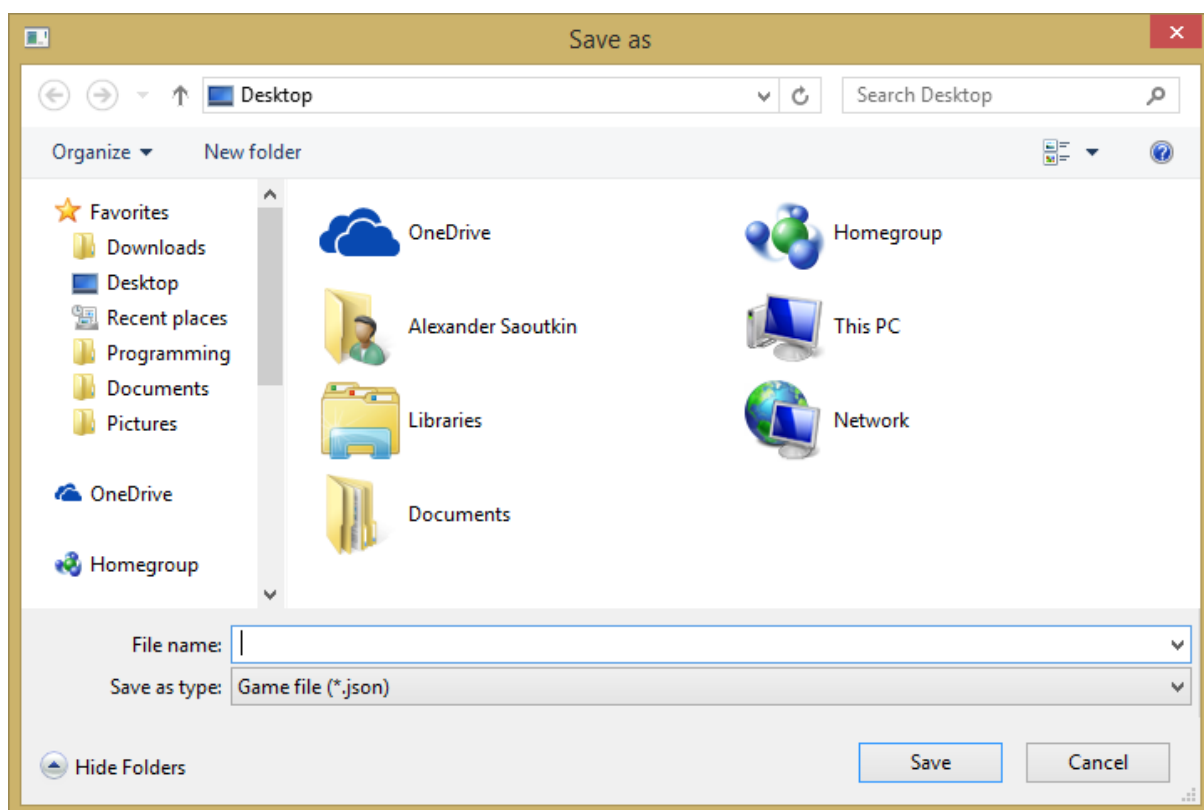
Figure A.18: Test 19 - "Save as" dialog to save game.
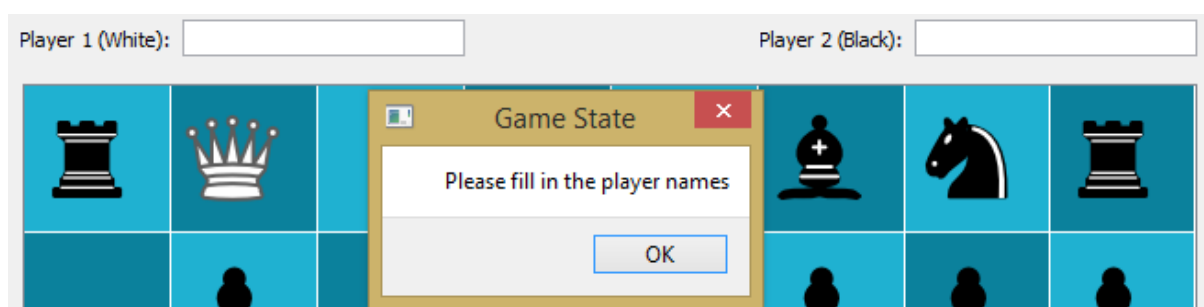
Figure A.19: Test 20 - check that game saved only when required data is filled.
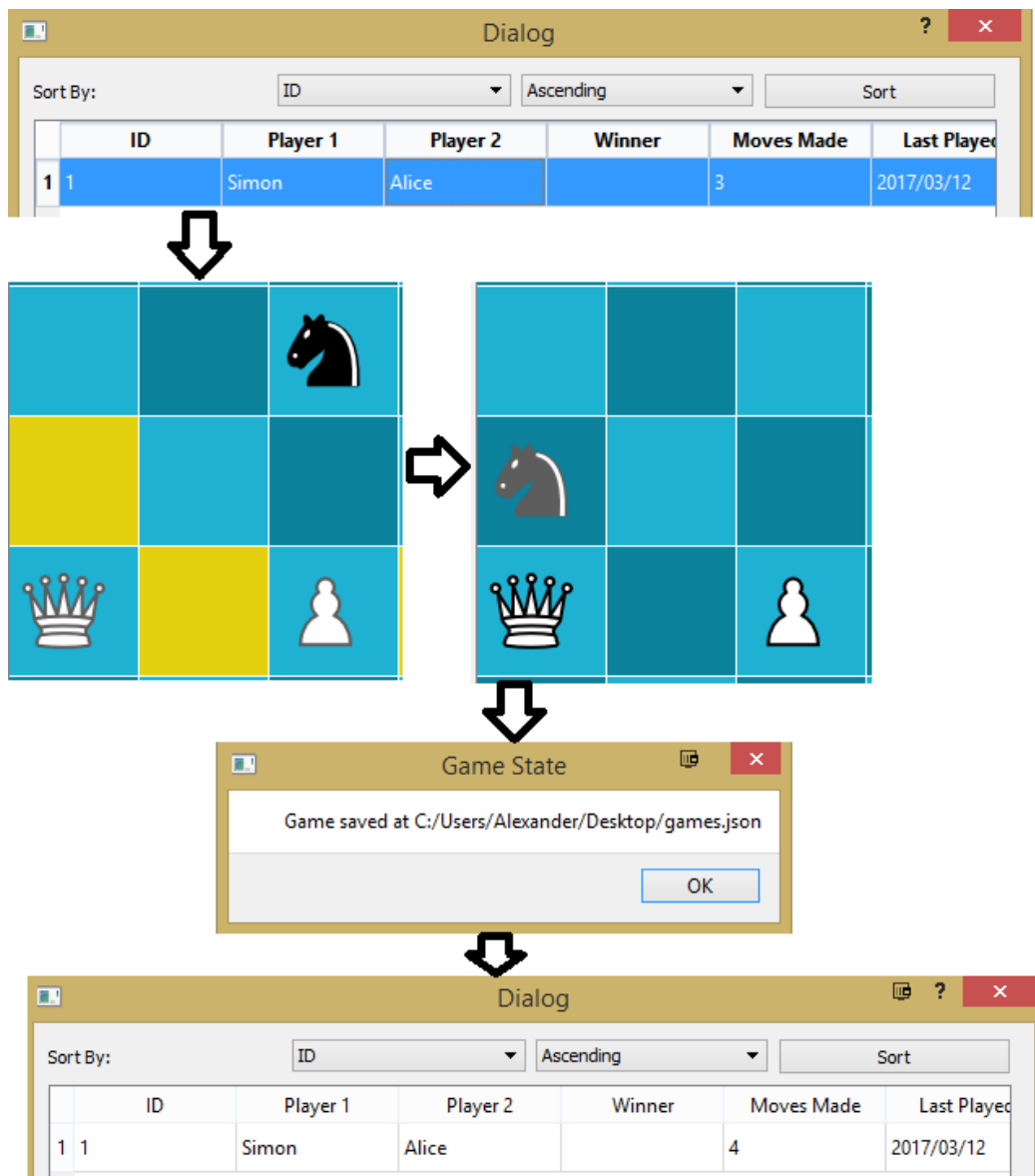
Figure A.20: Test 21 - check that game saved only when required data is filled.
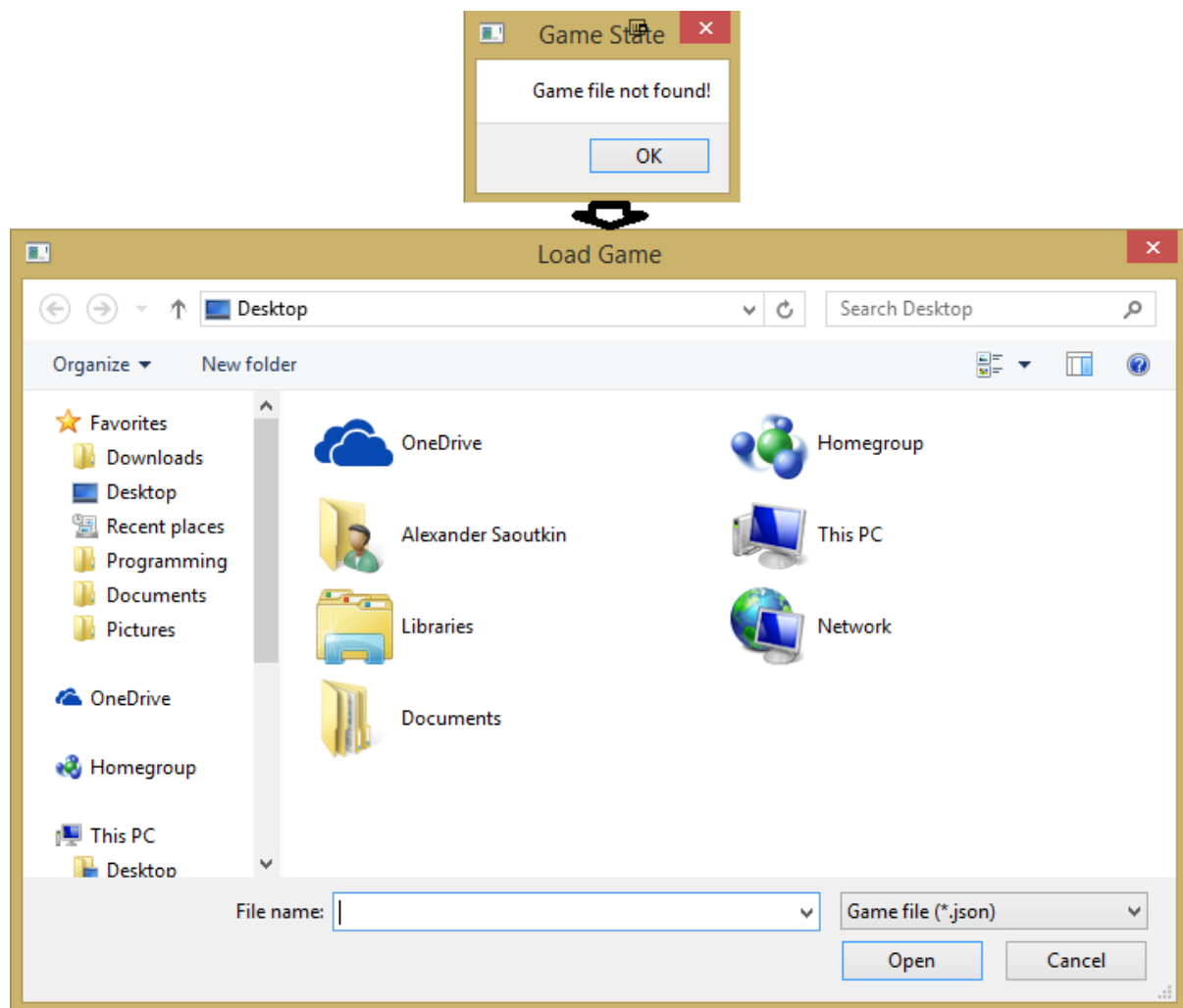
Figure A.21: Test 22 - check that game saved only when required data is filled.
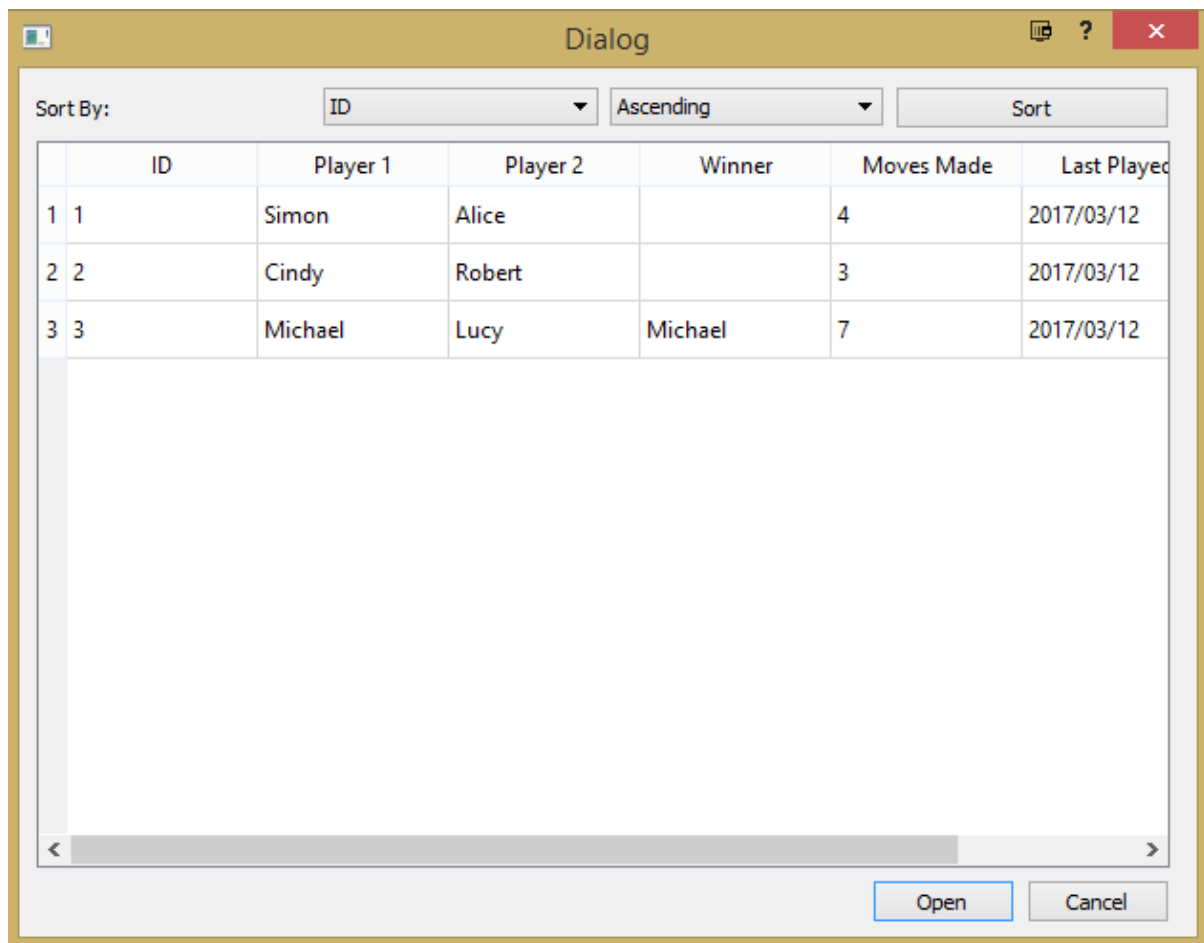
Figure A.22: Test 23 - check that game saved only when required data is filled.
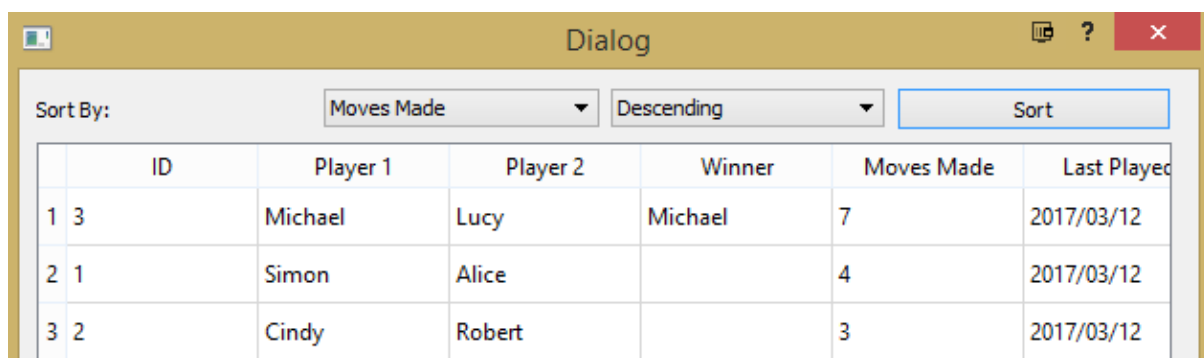
Figure A.23: Test 24 - check that game saved only when required data is filled.

A.2   Source Code

board.py

```python
from pieces import *
import copy
__author__ = "Alexander Saoutkin"


class Board(object):
    """Class which manages the pieces on the board.

    This is the model for the ChessBoardController class.
    The primary function of this class is to allow the controller to determine
    the legal moves a given piece can make and to move pieces on the board, whilst
    performing checks of the game state after every move.

    Attributes:
        id (int): the unique identifier for a particular game.
        last_played (str): The date at which the game was last saved.
        turn (str): The colour of the current player's turn.
        move_num (int): how many moves have been made in the game.
        winner (str): The name of the winner.
        colour_in_check (str): if a player is in check, their colour is held in this
    variable.
        is_stalemate (bool): shows if game is in stalemate or not.
        game_over (bool): shows if game is over or not.
        must_promote (bool): true if a player must promote their pawn, false
    otherwise.
        enpassant_possible (dict): shows if black and white can complete an en
    passant move.
        enpassant_move (dict): stores the en passant move for both colours if they
    exist.
        player_one (str): stores the name of the first player (white).
        player_two (str): stores the name of the second player (black).
        board (list): an 8*8 2D list which maps to pieces on the board.
    """

    def __init__(self, game=None):
        """Loads a game if given one, otherwise intialises a new game.

        Args:
            game (dict): contains all data needed to load a game into a Board object.
        """
        if game is not None:
            self.id = game['id']
            self.last_played = game['last_played']
            self.turn = game['turn']
            self.move_num = game['move_num']
            self.winner = game['winner']
            self.colour_in_check = game['colour_in_check']
            self.is_stalemate = game['is_stalemate']
            self.game_over = game['game_over']
            self.must_promote = game['must_promote']
            self.enpassant_possible = {
```

```python
48                  'Black': game['enpassant_possible']['Black'],
49                  'White': game['enpassant_possible']['White']
50              }
51          self.enpassant_move = {
52                  'from': game['enpassant_move']['from'],
53                  'to': game['enpassant_move']['to'],
54                  'taken': game['enpassant_move']['taken']
55              }
56          self.player_one = game['player_one']
57          self.player_two = game['player_two']
58          self.board = [[0 for x in range(8)] for y in range(8)]
59          for piece in game['pieces']['queens']:
60              self.board[piece['position'][0]][piece['position'][1]] = \
61                  Queen(piece['position'], piece['colour'])
62          for piece in game['pieces']['knights']:
63              self.board[piece['position'][0]][piece['position'][1]] = \
64                  Knight(piece['position'], piece['colour'])
65          for piece in game['pieces']['bishops']:
66              self.board[piece['position'][0]][piece['position'][1]] = \
67                  Bishop(piece['position'], piece['colour'])
68          for piece in game['pieces']['kings']:
69              self.board[piece['position'][0]][piece['position'][1]] = \
70                  King(piece['position'],
71                      piece['colour'],piece['has_moved'],['castling_moves'])
72          for piece in game['pieces']['pawns']:
73              self.board[piece['position'][0]][piece['position'][1]] = \
74                  Pawn(piece['position'], piece['colour'],piece['first_moved'])
75          for piece in game['pieces']['rooks']:
76              self.board[piece['position'][0]][piece['position'][1]] = \
77                  Rook(piece['position'], piece['colour'],piece['has_moved'])
78      else:
79          self.id = None
80          self.last_played = None
81          self.turn = "White"
82          self.move_num = 1
83          self.winner = ""
84          self.colour_in_check = ""
85          self.is_stalemate = False
86          self.game_over = False
87          self.must_promote = False
88          self.enpassant_possible = {
89              'Black': True,
90              'White': True
91              }
92          self.enpassant_move = {
93              'from': [],
94              'to': [],
95              'taken': []
96              }
97          self.player_one = ""
98          self.player_two = ""
99          self.new_board()
100
101  def new_board(self):
102      """Creates a new board."""
103      self.board = []
104      # Black side of the board
105      row = [
```

```
106              Rook([0, 0], "Black"),
107              Knight([0, 1], "Black"),
108              Bishop([0, 2], "Black"),
109              Queen([0, 3], "Black"),
110              King([0, 4], "Black"),
111              Bishop([0, 5], "Black"),
112              Knight([0, 6], "Black"),
113              Rook([0, 7], "Black")
114          ]
115          self.board.append(row)
116          row = [
117              Pawn([1, 0], "Black"),
118              Pawn([1, 1], "Black"),
119              Pawn([1, 2], "Black"),
120              Pawn([1, 3], "Black"),
121              Pawn([1, 4], "Black"),
122              Pawn([1, 5], "Black"),
123              Pawn([1, 6], "Black"),
124              Pawn([1, 7], "Black")
125          ]
126          self.board.append(row)
127          # Add empty space
128          for i in range(4):
129              row = [0 for x in range(8)]
130              self.board.append(row)
131          # Add white pieces
132          row = [
133              Pawn([6, 0], "White"),
134              Pawn([6, 1], "White"),
135              Pawn([6, 2], "White"),
136              Pawn([6, 3], "White"),
137              Pawn([6, 4], "White"),
138              Pawn([6, 5], "White"),
139              Pawn([6, 6], "White"),
140              Pawn([6, 7], "White")
141          ]
142          self.board.append(row)
143          row = [
144              Rook([7, 0], "White"),
145              Knight([7, 1], "White"),
146              Bishop([7, 2], "White"),
147              Queen([7, 3], "White"),
148              King([7, 4], "White"),
149              Bishop([7, 5], "White"),
150              Knight([7, 6], "White"),
151              Rook([7, 7], "White")
152          ]
153          self.board.append(row)
154
155      def preliminary_move_piece(self, chess_board, old_coords, new_coords):
156          """Will move a piece temporarily. e.g. used to see if piece puts itself in
              ↪    check.
157
158          Args:
159              chess_board (list): the board to be used to move the piece.
160              old_coords (list): coordinates of the piece to be moved.
161              new_coords (list): coordinates of where the piece will be moved to.
162
```

```python
163                 Returns:
164                     list: returns a chess board with the piece moved.
165                 """
166             chess_board[new_coords[0]][new_coords[1]] =
        ↪   copy.deepcopy(chess_board[old_coords[0]][old_coords[1]])
167             chess_board[old_coords[0]][old_coords[1]] = 0
168             if isinstance(chess_board[new_coords[0]][new_coords[1]], Piece):
169                 chess_board[new_coords[0]][new_coords[1]].position = [new_coords[0],
            ↪   new_coords[1]]
170                 chess_board[new_coords[0]][new_coords[1]].calculate_possible_moves()
171             return chess_board
172
173         def preliminary_enpassant(self, chess_board, old_coords, new_coords,
        ↪   removed_coords):
174             """Will perform enpassant temporarily e.g. used to see if piece puts itself
            ↪   in check.
175
176             Args:
177                 chess_board (list): the board to be used to move the piece.
178                 old_coords (list): coordinates of the piece to be moved.
179                 new_coords (list): coordinates of where the piece will be moved to.
180                 removed_coords (list): coordinates of the piece to be taken.
181
182             Returns:
183                 list: returns a chess board with the piece moved.
184             """
185             chess_board[new_coords[0]][new_coords[1]] =
        ↪   copy.deepcopy(chess_board[old_coords[0]][old_coords[1]])
186             chess_board[old_coords[0]][old_coords[1]] = 0
187             chess_board[removed_coords[0]][removed_coords[1]] = 0
188             if isinstance(chess_board[new_coords[0]][new_coords[1]], Piece):
189                 chess_board[new_coords[0]][new_coords[1]].position = [new_coords[0],
            ↪   new_coords[1]]
190                 chess_board[new_coords[0]][new_coords[1]].calculate_possible_moves()
191             return chess_board
192
193         def permanently_move_piece(self, chess_board, old_coords, new_coords):
194             """
195             Moves a piece on the board permanently.
196             It will also check for change in game state (checkmate, check, stalemate).
197
198             Args:
199                 chess_board (list): the board to be used to move the piece.
200                 old_coords (list): coordinates of the piece to be moved.
201                 new_coords (list): coordinates of where the piece will be moved to.
202
203             Returns:
204                 list: returns a chess board with the piece moved.
205             """
206             if not self.game_over:
207                 # Performs enpassant if conditions are met.
208                 if old_coords == self.enpassant_move['from'] and new_coords ==
                ↪   self.enpassant_move['to'] and self.enpassant_possible[self.turn]:
209                     chess_board[self.enpassant_move['to'][0]][self.enpassant_move['to'][1 ⌄
                    ↪   ]] =
                    ↪   copy.deepcopy(chess_board[old_coords[0]][old_coords[1]])
210                     chess_board[old_coords[0]][old_coords[1]] = 0
```

```python
211             chess_board[self.enpassant_move['taken'][0]][self.enpassant_move['tak
        ↪   en'][1]] =
        ↪   0
212             if isinstance(chess_board[new_coords[0]][new_coords[1]], Piece):
213                 chess_board[new_coords[0]][new_coords[1]].position =
            ↪   [new_coords[0], new_coords[1]]
214                 chess_board[new_coords[0]][new_coords[1]].calculate_possible_move
            ↪   s()
215             self.enpassant_possible[self.turn] = False
216             self.enpassant_move['from'] = []
217             self.enpassant_move['to'] = []
218             self.enpassant_move['taken'] = []
219         elif isinstance(chess_board[old_coords[0]][old_coords[1]], King):
220             chess_board[new_coords[0]][new_coords[1]] =
        ↪   copy.deepcopy(self.board[old_coords[0]][old_coords[1]])
221             chess_board[old_coords[0]][old_coords[1]] = 0
222             # If legal move is castling move, perform castling...
223             if new_coords in
        ↪   chess_board[new_coords[0]][new_coords[1]].castling_moves:
224                 if old_coords[0] == 0:
225                     if new_coords[1] == 2:
226                         chess_board[0][3] = copy.deepcopy(self.board[0][0])
227                         chess_board[0][3].position = [0, 3]
228                         chess_board[0][3].calculate_possible_moves()
229                         chess_board[0][0] = 0
230                     else:
231                         chess_board[0][5] = copy.deepcopy(self.board[0][7])
232                         chess_board[0][5].position = [0, 5]
233                         chess_board[0][5].calculate_possible_moves()
234                         chess_board[0][7] = 0
235                 else:
236                     if new_coords[1] == 2:
237                         chess_board[7][3] = copy.deepcopy(self.board[7][0])
238                         chess_board[7][3].position = [7, 3]
239                         chess_board[7][3].calculate_possible_moves()
240                         chess_board[7][0] = 0
241                     else:
242                         chess_board[7][5] = copy.deepcopy(self.board[7][7])
243                         chess_board[7][5].position = [7, 5]
244                         chess_board[7][5].calculate_possible_moves()
245                         chess_board[7][7] = 0
246         else:
247             # if enpassant possible but not done, then can never be done again.
248             if self.enpassant_move['from'] and self.enpassant_move['to']:
249                 self.enpassant_possible[self.turn] = False
250                 self.enpassant_move['from'] = []
251                 self.enpassant_move['to'] = []
252                 self.enpassant_move['taken'] = []
253             chess_board[new_coords[0]][new_coords[1]] =
        ↪   copy.deepcopy(self.board[old_coords[0]][old_coords[1]])
254             chess_board[old_coords[0]][old_coords[1]] = 0
255         if isinstance(chess_board[new_coords[0]][new_coords[1]], Pawn):
256             chess_board[new_coords[0]][new_coords[1]].first_moved = self.move_num
257             if chess_board[new_coords[0]][new_coords[1]].colour == "Black" and
        ↪   new_coords[0] == 7:
258                 self.must_promote = True
259             elif chess_board[new_coords[0]][new_coords[1]].colour == "White" and
        ↪   new_coords[0] == 0:
```

65

```
260                    self.must_promote = True
261
262                elif isinstance(chess_board[new_coords[0]][new_coords[1]], Rook) or
       ↪   isinstance(chess_board[new_coords[0]][new_coords[1]], King):
263                    # Won't work for rook when castling, but that doesn't matter as king
       ↪   has already moved, so can't castle anyway
264                    chess_board[new_coords[0]][new_coords[1]].has_moved = True
265            self.move_num += 1
266            if self.move_num % 2 == 0:
267                self.turn = "Black"
268            else:
269                self.turn = "White"
270            if isinstance(chess_board[new_coords[0]][new_coords[1]], Piece):
271                chess_board[new_coords[0]][new_coords[1]].position = [new_coords[0],
       ↪   new_coords[1]]
272                chess_board[new_coords[0]][new_coords[1]].calculate_possible_moves()
273            return chess_board
274
275    def permanently_promote_piece(self, type, coords):
276        """Promotes a pawn to a piece of a certain type.
277
278        Args:
279            type (str): Name of the type of piece to promote to.
280            coords (list): coordinates of the pawn to be promoted.
281        """
282        colour = self.board[coords[0]][coords[1]].colour
283        if type == "Queen":
284            self.board[coords[0]][coords[1]] = Queen(coords, colour)
285            self.board[coords[0]][coords[1]].position = coords
286            self.board[coords[0]][coords[1]].calculate_possible_moves()
287        elif type == "Knight":
288            self.board[coords[0]][coords[1]] = Knight(coords, colour)
289            self.board[coords[0]][coords[1]].position = coords
290            self.board[coords[0]][coords[1]].calculate_possible_moves()
291        elif type == "Rook":
292            self.board[coords[0]][coords[1]] = Rook(coords, colour)
293            self.board[coords[0]][coords[1]].position = coords
294            self.board[coords[0]][coords[1]].calculate_possible_moves()
295        elif type == "Bishop":
296            self.board[coords[0]][coords[1]] = Bishop(coords, colour)
297            self.board[coords[0]][coords[1]].position = coords
298            self.board[coords[0]][coords[1]].calculate_possible_moves()
299        self.must_promote = False
300
301    def check_game_state(self):
302        """Completes a check of the state of the game."""
303        if self.calculate_is_checkmate(self.turn, self.board):
304            self.game_over = True
305            if self.turn == "Black":
306                if self.player_one:
307                    self.winner = self.player_one
308                else:
309                    self.winner = "White"
310            else:
311                if self.player_two:
312                    self.winner = self.player_two
313                else:
314                    self.winner = "Black"
```

```python
315                elif self.is_in_check(self.turn, self.board):
316                    self.colour_in_check = self.turn
317                elif self.calculate_is_stalemate(self.board):
318                    self.is_stalemate = True
319                    self.game_over = True
320                else:
321                    self.colour_in_check = ""


324        def get_king_coords(self, colour, board):
325            """Finds the coords of the king depending on its colour.
326
327            Args:
328                colour (str): color of the king to find.
329                board (list): the board to find the kind from.
330
331            Returns:
332                list: coordinates of the king in the form [x,y] (0-based).
333            """
334            for x in range(8):
335                for y in range(8):
336                    if isinstance(board[x][y], King) and board[x][y].colour == colour:
337                        return [x, y]
338
339        def calculate_legal_moves(self, piece):
340            """Gets all the legal moves of a piece and returns them.
341
342            Args:
343                piece (Piece): The piece to calculate the legal moves for.
344
345            Returns:
346                list: A list of all the legal moves a piece can make.
347            """
348            legal_moves = []
349            original_board = copy.deepcopy(self.board) # fixed the moving bug.
350            possible_legal_moves = self.get_attacking_moves(piece, original_board)
351            if not isinstance(piece, Pawn):
352                for move in possible_legal_moves:
353                    original_board = copy.deepcopy(self.board)
354                    if isinstance(original_board[move[0]][move[1]], Piece) and not
                        ↪  isinstance(original_board[move[0]][move[1]], King):
355                        if not piece.colour == original_board[move[0]][move[1]].colour:
356                            possible_board = self.preliminary_move_piece(original_board,
                                ↪  piece.position, move)
357                            if not self.is_in_check(piece.colour, possible_board):
358                                legal_moves.append(move)
359                    elif not isinstance(original_board[move[0]][move[1]], King):
360                        possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, move)
361                        if not self.is_in_check(piece.colour, possible_board):
362                            legal_moves.append(move)
363                if isinstance(piece, King):
364                    original_board = copy.deepcopy(self.board)
365                    self.get_castling_moves(piece, original_board)
366                    if piece.castling_moves:
367                        for move in piece.castling_moves:
368                            legal_moves.append(move)
369            else:
```

```python
370                 # vertical legal moves for pawn
371                 if piece.colour == "White":
372                     if piece.position[0] > 0 and not isinstance(original_board[piece.posi
                        ↪  tion[0]-1][piece.position[1]],
                        ↪  Piece):
373                         possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, [piece.position[0]-1, piece.position[1]])
374                         if not self.is_in_check(piece.colour, possible_board):
375                             legal_moves.append([piece.position[0]-1, piece.position[1]])
376                     original_board = copy.deepcopy(self.board)
377                     if piece.position[0] == 6 and not isinstance(original_board[piece.pos
                        ↪  ition[0]-2][piece.position[1]],
                        ↪  Piece):
378                         possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, [piece.position[0]-2, piece.position[1]])
379                         if not (self.is_in_check(piece.colour, possible_board) or isinsta
                            ↪  nce(original_board[piece.position[0]-1][piece.position[1]],
                            ↪  Piece)):
380                             legal_moves.append([piece.position[0]-2, piece.position[1]])
381                 else:
382                     if piece.position[0] < 7 and not isinstance(original_board[piece.posi
                        ↪  tion[0]+1][piece.position[1]],
                        ↪  Piece):
383                         possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, [piece.position[0]+1, piece.position[1]])
384                         if not self.is_in_check(piece.colour, possible_board):
385                             legal_moves.append([piece.position[0]+1, piece.position[1]])
386                     original_board = copy.deepcopy(self.board)
387                     if piece.position[0] == 1 and not isinstance(original_board[piece.pos
                        ↪  ition[0]+2][piece.position[1]],
                        ↪  Piece):
388                         possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, [piece.position[0]+2, piece.position[1]])
389                         if not(self.is_in_check(piece.colour, possible_board) or isinstan
                            ↪  ce(original_board[piece.position[0]+1][piece.position[1]],
                            ↪  Piece)):
390                             legal_moves.append([piece.position[0]+2, piece.position[1]])
391                 # When pawn moves diagonally to take piece
392                 for move in possible_legal_moves:
393                     original_board = copy.deepcopy(self.board)
394                     if isinstance(original_board[move[0]][move[1]], Piece) and not
                        ↪  isinstance(original_board[move[0]][move[1]], King):
395                         possible_board = self.preliminary_move_piece(original_board,
                            ↪  piece.position, move)
396                         if not self.is_in_check(piece.colour, possible_board):
397                             legal_moves.append(move)
398                 # If enpassant is possible, adds as a possible move.
399                 self.can_enpassant(piece, original_board)
400                 if self.enpassant_move['to']:
401                         legal_moves.append(self.enpassant_move['to'])
402         return legal_moves
403
404     def get_attacking_moves(self, piece, board):
405         """
406         Get all the attacking moves of a piece and return them.
407         The difference between this function and get_legal_moves() is that
408         this shows you what pieces can force a check while the other functions
409         tells you whether the piece can move there.
```

```python
410
411         Args:
412             piece (Piece): The piece to calculate the legal moves for.
413             board (board): board used to determine attacking moves.
414
415         Returns:
416             list: A list of all attacking moves a piece can make.
417         """
418         legal_moves = []
419         illegal_moves = []
420         illegal_moves.append(piece.position)
421         def get_legal_moves():
422             for move in piece.possible_moves:
423                 if move not in illegal_moves:
424                     legal_moves.append(move)
425             return legal_moves
426         if isinstance(piece, Rook):
427             for move in piece.possible_moves:
428                 # If there is a piece in the way
429                 if board[move[0]][move[1]]:
430                     # If piece in the way is above
431                     if piece.position[0] > move[0]:
432                         # Then it cannot influence anything above that piece
433                         for i in range(move[0]):
434                             if [i, move[1]] not in illegal_moves:
435                                 illegal_moves.append([i, move[1]])
436                     # If piece in the way is below
437                     elif piece.position[0] < move[0]:
438                         # Then it cannot influence anything below that piece
439                         for i in range(7, move[0], -1):
440                             if [i, move[1]] not in illegal_moves:
441                                 illegal_moves.append([i, move[1]])
442                     # If piece in the way is to the left
443                     elif piece.position[1] > move[1]:
444                         # Then it cannot influence anything to the left
445                         for i in range(move[1]):
446                             if [move[0], i] not in illegal_moves:
447                                 illegal_moves.append([move[0], i])
448                     # If piece in the way is to the right
449                     elif piece.position[1] < move[1]:
450                         # Then it cannot influence anything to the right
451                         for i in range(7, move[1], -1):
452                             if [move[0], i] not in illegal_moves:
453                                 illegal_moves.append([move[0], i])
454             return get_legal_moves()
455         elif isinstance(piece, Bishop):
456             for move in piece.possible_moves:
457                 if board[move[0]][move[1]]:
458                     counter = 0
459                     # If piece in the way is north-west
460                     if piece.position[0] > move[0] and piece.position[1] > move[1]:
461                         # Then it cannot influence anything north-west
462                         while move[0]-counter != 0 and move[1]-counter != 0:
463                             counter += 1
464                             if [move[0]-counter, move[1]-counter] not in
                                 ↪  illegal_moves:
465                                 illegal_moves.append([move[0]-counter,
                                     ↪  move[1]-counter])
```

69

```python
466                         # If piece in the way is north-east
467                         elif piece.position[0] > move[0] and piece.position[1] < move[1]:
468                             # Then it cannot influence anything north-east
469                             while move[0]-counter != 0 and move[1]+counter != 7:
470                                 counter += 1
471                                 if [move[0]-counter, move[1]+counter] not in
                                 ↪   illegal_moves:
472                                     illegal_moves.append([move[0]-counter,
                                     ↪   move[1]+counter])
473                         # If piece in the way is south-west
474                         elif piece.position[0] < move[0] and piece.position[1] > move[1]:
475                             # Then it cannot influence anything south-west
476                             while move[0]+counter != 7 and move[1]-counter != 0:
477                                 counter += 1
478                                 if [move[0]+counter, move[1]-counter] not in
                                 ↪   illegal_moves:
479                                     illegal_moves.append([move[0]+counter,
                                     ↪   move[1]-counter])
480                         # If piece in the way is south-east
481                         elif piece.position[0] < move[0] and piece.position[1] < move[1]:
482                             # Then it cannot influence anything south-east
483                             while move[0]+counter != 7 and move[1]+counter != 7:
484                                 counter += 1
485                                 if [move[0]+counter, move[1]+counter] not in
                                 ↪   illegal_moves:
486                                     illegal_moves.append([move[0]+counter,
                                     ↪   move[1]+counter])
487             return get_legal_moves()
488         # This function is a combination of Rook and Bishop
489         elif isinstance(piece, Queen):
490             for move in piece.possible_moves:
491                 if board[move[0]][move[1]]:
492                     counter = 0
493                     # If piece in the way is above
494                     if piece.position[0] > move[0] and piece.position[1] == move[1]:
495                         # Then it cannot influence anything above it
496                         for i in range(move[0]):
497                             if [i, move[1]] not in illegal_moves:
498                                 illegal_moves.append([i, move[1]])
499                     # If piece in the way is below
500                     elif piece.position[0] < move[0] and piece.position[1] == move[1]:
501                         # Then it cannot influence below it...
502                         for i in range(7, move[0], -1):
503                             if [i, move[1]] not in illegal_moves:
504                                 illegal_moves.append([i, move[1]])
505                     # If piece in the way is to the left
506                     elif piece.position[1] > move[1] and piece.position[0] == move[0]:
507                         # Then it cannot influence anything to the left of it
508                         for i in range(move[1]):
509                             if [move[0], i] not in illegal_moves:
510                                 illegal_moves.append([move[0], i])
511                     # If piece in the way is to the right
512                     elif piece.position[1] < move[1] and piece.position[0] == move[0]:
513                         # Then it cannot influence anything to the right of it
514                         for i in range(7, move[1], -1):
515                             if [move[0], i] not in illegal_moves:
516                                 illegal_moves.append([move[0], i])
517                     # If piece in the way is north-west
```

```python
518              elif piece.position[0] > move[0] and piece.position[1] > move[1]:
519                  # Then it cannot influence anything north-west
520                  while move[0]-counter != 0 and move[1]-counter != 0:
521                      counter += 1
522                      if [move[0]-counter, move[1]-counter] not in
                          ↪  illegal_moves:
523                          illegal_moves.append([move[0]-counter,
                              ↪  move[1]-counter])
524              # If piece in the way is north-east
525              elif piece.position[0] > move[0] and piece.position[1] < move[1]:
526                  # Then it cannot influence anything north-east
527                  while move[0]-counter != 0 and move[1]+counter != 7:
528                      counter += 1
529                      if [move[0]-counter, move[1]+counter] not in
                          ↪  illegal_moves:
530                          illegal_moves.append([move[0]-counter,
                              ↪  move[1]+counter])
531              # If piece in the way is south-west
532              elif piece.position[0] < move[0] and piece.position[1] > move[1]:
533                  # Then it cannot influence anything south-west
534                  while move[0]+counter != 7 and move[1]-counter != 0:
535                      counter += 1
536                      if [move[0]+counter, move[1]-counter] not in
                          ↪  illegal_moves:
537                          illegal_moves.append([move[0]+counter,
                              ↪  move[1]-counter])
538              # If piece in the way is south-east
539              elif piece.position[0] < move[0] and piece.position[1] < move[1]:
540                  # Then it cannot influence anything south-east
541                  while move[0]+counter != 7 and move[1]+counter != 7:
542                      counter += 1
543                      if [move[0]+counter, move[1]+counter] not in
                          ↪  illegal_moves:
544                          illegal_moves.append([move[0]+counter,
                              ↪  move[1]+counter])
545          return get_legal_moves()
546      elif isinstance(piece, Pawn):
547          if piece.colour == "White":
548              if piece.position[0] > 0 and piece.position[1] > 0:
549                  if isinstance(board[piece.position[0]-1][piece.position[1]-1],
                      ↪  Piece):
550                      if board[piece.position[0]-1][piece.position[1]-1].colour ==
                          ↪  "Black":
551                          legal_moves.append([piece.position[0]-1,
                              ↪  piece.position[1]-1])
552              if piece.position[0] > 0 and piece.position[1] < 7:
553                  if isinstance(board[piece.position[0]-1][piece.position[1]+1],
                      ↪  Piece):
554                      if board[piece.position[0]-1][piece.position[1]+1].colour ==
                          ↪  "Black":
555                          legal_moves.append([piece.position[0]-1,
                              ↪  piece.position[1]+1])
556          elif piece.colour == "Black":
557              if piece.position[0] < 7 and piece.position[1] < 7:
558                  if isinstance(board[piece.position[0]+1][piece.position[1]+1],
                      ↪  Piece):
559                      if board[piece.position[0]+1][piece.position[1]+1].colour ==
                          ↪  "White":
```

71

```python
560                     legal_moves.append([piece.position[0]+1,
                        ↪  piece.position[1]+1])
561                 if piece.position[0] < 7 and piece.position[1] > 0:
562                     if isinstance(board[piece.position[0]+1][piece.position[1]-1],
                        ↪  Piece):
563                         if board[piece.position[0]+1][piece.position[1]-1].colour ==
                        ↪  "White":
564                             legal_moves.append([piece.position[0]+1,
                        ↪  piece.position[1]-1])
565             return legal_moves
566         elif isinstance(piece, Knight):
567             return piece.possible_moves
568         elif isinstance(piece, King):
569             return piece.possible_moves
570         else:
571             return legal_moves
572
573     def calculate_is_checkmate(self, colour, board):
574         """Finds out if a player is in checkmate.
575
576         Args:
577             colour (str): Colour that we are checking for if they are in checkmate.
578             board (list): the board of the game being played.
579
580         Returns:
581             bool: True if the game is in a state of checkmate, False otherwise.
582         """
583         if self.is_in_check(colour, board):
584             for row in board:
585                 for piece in row:
586                     if isinstance(piece, Piece) and piece.colour == colour:
587                         if self.calculate_legal_moves(piece):
588                             return False
589             return True
590         else:
591             return False
592
593     def calculate_is_stalemate(self, board):
594         """Finds out if a game is in stalemate.
595
596         Args:
597             board (list): the board of the game being played.
598
599         Returns:
600             bool: True if the game is in a state of stalemate, False otherwise.
601         """
602         if not self.is_in_check("White", board) and not self.is_in_check("Black",
            ↪  board):
603             for row in board:
604                 for piece in row:
605                     if isinstance(piece, Piece) and piece.colour == self.turn:
606                         if self.calculate_legal_moves(piece):
607                             return False
608             return True
609         else:
610             return False
611
612     def is_in_check(self, colour, possible_board):
```

```
613            """Checks if the king of the corresponding colour is in check.
614
615            Args:
616                colour (str): colour that we are checking if they are in check.
617                possible_board (list): the board of the game being played.
618
619            Returns:
620                bool: True if the player is in check, False otherwise.
621            """
622            king_coords = self.get_king_coords(colour, possible_board)
623            for x in range(8):
624                for y in range(8):
625                    if isinstance(possible_board[x][y], Piece) and
                        ↪  possible_board[x][y].colour != colour:
626                        if king_coords in self.get_attacking_moves(possible_board[x][y],
                            ↪  possible_board):
627                            return True
628            return False
629
630        def get_castling_moves(self, king, original_board):
631            """Finds castling moves, if they exist, for a given king.
632
633            Args:
634                king (King): the king that we are finding castling moves for.
635                original_board (list): the board of the game being played.
636            """
637            king.castling_moves = []
638            if not king.has_moved and not self.is_in_check(king.colour, self.board):
639                if king.colour == "Black":
640                    if isinstance(original_board[0][0], Rook):
641                        if original_board[0][0].colour == "Black" and not
                            ↪  original_board[0][0].has_moved and not(original_board[0][2]
                            ↪  or original_board[0][3]):
642                            original_board = copy.deepcopy(self.board)
643                            possible_board = self.preliminary_move_piece(original_board,
                                ↪  king.position, [king.position[0], king.position[1]-1])
644                            if not self.is_in_check(king.colour, possible_board):
645                                original_board = copy.deepcopy(self.board)
646                                possible_board =
                                    ↪  self.preliminary_move_piece(original_board,
                                    ↪  king.position, [king.position[0],
                                    ↪  king.position[1]-2])
647                                if not self.is_in_check(king.colour, possible_board):
648                                    king.castling_moves.append([king.position[0],
                                        ↪  king.position[1]-2])
649                    if isinstance(original_board[0][7], Rook):
650                        if original_board[0][7].colour == "Black" and not
                            ↪  original_board[0][7].has_moved and not(original_board[0][5]
                            ↪  or original_board[0][6]):
651                            original_board = copy.deepcopy(self.board)
652                            possible_board = self.preliminary_move_piece(original_board,
                                ↪  king.position, [king.position[0], king.position[1]+1])
653                            if not self.is_in_check(king.colour, possible_board):
654                                original_board = copy.deepcopy(self.board)
655                                possible_board =
                                    ↪  self.preliminary_move_piece(original_board,
                                    ↪  king.position, [king.position[0],
                                    ↪  king.position[1]+2])
```

```python
656                         if not self.is_in_check(king.colour, possible_board):
657                             king.castling_moves.append([king.position[0],
                             ↪   king.position[1]+2])
658                 else:
659                     if isinstance(original_board[7][0], Rook):
660                         if original_board[7][0].colour == "White" and not
                         ↪   original_board[7][0].has_moved and not(original_board[7][2]
                         ↪   or original_board[7][3]):
661                             original_board = copy.deepcopy(self.board)
662                             possible_board = self.preliminary_move_piece(original_board,
                             ↪   king.position, [king.position[0], king.position[1]-1])
663                             if not self.is_in_check(king.colour, possible_board):
664                                 original_board = copy.deepcopy(self.board)
665                                 possible_board =
                                 ↪   self.preliminary_move_piece(original_board,
                                 ↪   king.position, [king.position[0],
                                 ↪   king.position[1]-2])
666                                 if not self.is_in_check(king.colour, possible_board):
667                                     king.castling_moves.append([king.position[0],
                                     ↪   king.position[1]-2])
668                     if isinstance(original_board[7][7], Rook):
669                         if original_board[7][7].colour == "White" and not
                         ↪   original_board[7][7].has_moved and not(original_board[7][5]
                         ↪   or original_board[7][6]):
670                             original_board = copy.deepcopy(self.board)
671                             possible_board = self.preliminary_move_piece(original_board,
                             ↪   king.position, [king.position[0], king.position[1]+1])
672                             if not self.is_in_check(king.colour, possible_board):
673                                 original_board = copy.deepcopy(self.board)
674                                 possible_board =
                                 ↪   self.preliminary_move_piece(original_board,
                                 ↪   king.position, [king.position[0],
                                 ↪   king.position[1]+2])
675                                 if not self.is_in_check(king.colour, possible_board):
676                                     king.castling_moves.append([king.position[0],
                                     ↪   king.position[1]+2])
677
678     def can_enpassant(self, pawn, possible_board):
679         """Checks if en passant is possible, and assigns the en passant move if so.
680
681         Args:
682             pawn (Pawn): the pawn that we are determining if it can perform en
    ↪   passant.
683             possible_board (list): the board of the game being played.
684
685         Raises:
686             IndexError: raised when index is -1 or 8 as these indices do not exist in
    ↪   possible_board.
687         """
688         if self.enpassant_possible[pawn.colour] and ((pawn.position[0] == 4 and
         ↪   pawn.colour == "Black") or (pawn.position[0] == 3 and pawn.colour ==
         ↪   "White")):
689             try:
690                 if isinstance(possible_board[pawn.position[0]][pawn.position[1]-1],
                     ↪   Pawn):
691                     if possible_board[pawn.position[0]][pawn.position[1]-1].first_mov⌐
                         ↪   ed == self.move_num -
                         ↪   1:
```

```python
692                    if pawn.colour == "Black":
693                        temp_board = self.preliminary_enpassant(possible_board,
                             ↪  pawn.position, [pawn.position[0] + 1,
                             ↪  pawn.position[1] - 1], [pawn.position[0],
                             ↪  pawn.position[1] - 1])
694                        if self.is_in_check("Black", temp_board):
695                            return False
696                        else:
697                            self.enpassant_move['from'] = pawn.position
698                            self.enpassant_move['to'] = [pawn.position[0] + 1,
                                 ↪  pawn.position[1] - 1]
699                            self.enpassant_move['taken'] = [pawn.position[0],
                                 ↪  pawn.position[1] - 1]
700                            return True
701                    else:
702                        temp_board = self.preliminary_enpassant(possible_board,
                             ↪  pawn.position, [pawn.position[0] - 1,
                             ↪  pawn.position[1] - 1], [pawn.position[0],
                             ↪  pawn.position[1] - 1])
703                        if self.is_in_check("White", temp_board):
704                            return False
705                        else:
706                            self.enpassant_move['from'] = pawn.position
707                            self.enpassant_move['to'] = [pawn.position[0] - 1,
                                 ↪  pawn.position[1] - 1]
708                            self.enpassant_move['taken'] = [pawn.position[0],
                                 ↪  pawn.position[1] - 1]
709                            return True
710                else:
711                    return False
712            except IndexError:
713                pass
714            try:
715                if isinstance(possible_board[pawn.position[0]][pawn.position[1]+1],
                     ↪  Pawn):
716                    if possible_board[pawn.position[0]][pawn.position[1]+1].first_mov⌋
                         ↪  ed == self.move_num -
                         ↪  1:
717                        if pawn.colour == "Black":
718                            temp_board = self.preliminary_enpassant(possible_board,
                                 ↪  pawn.position, [pawn.position[0] + 1,
                                 ↪  pawn.position[1] + 1], [pawn.position[0],
                                 ↪  pawn.position[1] + 1])
719                            if self.is_in_check("Black", temp_board):
720                                return False
721                            else:
722                                self.enpassant_move['from'] = pawn.position
723                                self.enpassant_move['to'] = [pawn.position[0] + 1,
                                     ↪  pawn.position[1] + 1]
724                                self.enpassant_move['taken'] = [pawn.position[0],
                                     ↪  pawn.position[1] + 1]
725                                return True
726                        else:
727                            temp_board = self.preliminary_enpassant(possible_board,
                                 ↪  pawn.position, [pawn.position[0] - 1,
                                 ↪  pawn.position[1] + 1], [pawn.position[0],
                                 ↪  pawn.position[1] + 1])
728                            if self.is_in_check("White", temp_board):
```

```
729                                    return False
730                                else:
731                                    self.enpassant_move['from'] = pawn.position
732                                    self.enpassant_move['to'] = [pawn.position[0] - 1,
                                    ↪  pawn.position[1] + 1]
733                                    self.enpassant_move['taken'] = [pawn.position[0],
                                    ↪  pawn.position[1] + 1]
734                                    return True
735                            else:
736                                return False
737                except IndexError:
738                    pass
739                return False
740            else:
741                return False
```

Chess.py

```
1   import sys
2   import controllers
3   from PySide import QtGui, QtCore
4
5   if __name__ == "__main__":
6       app = QtGui.QApplication(sys.argv)
7       win  = controllers.MainWindowController()
8
9       app.connect(app, QtCore.SIGNAL("lastWindowClosed()"), app, QtCore.SLOT("quit()"))
10      app.exec_()
```

controllers.py

```
1   import copy
2   import datetime
3   import json
4   import re
5   from builtins import IOError, FileNotFoundError, TypeError
6
7   from PySide import QtGui, QtCore
8
9   import algorithms
10  import views
11  from board import Board
12  from pieces import *
13
14
15  class MainWindowController(QtGui.QMainWindow, views.MainWindow):
16      """Controller for the main window of the application"""
17      def __init__(self):
18          super(MainWindowController, self).__init__()
19          self.setupUi(self)
20          self.setCentralWidget(ChessBoardController())
21          self.show()
22
```

```
23
24  class ChessBoardController(QtGui.QWidget, views.ChessBoard):
25      """Controller for the chess board.
26      Attributes pertaining to the views.ChessBoard class are not listed here as there
    ↪   are too many
27      and they merely refer to Qt GUI Objects.
28
29      Attributes:
30          board (Board): instance of the Board class which serves as the model for the
    ↪   controller.
31          settings (QSettings): a class used to store settings (namely file path) on
    ↪   the host computer.
32          from_cell (list): stores the coordinates of the cell that was chosen to move
    ↪   from.
33
34      """
35      def __init__(self):
36          """Initialises necessary variables, connects click events to methods and
             ↪   presents a new game to the user."""
37          super(ChessBoardController, self).__init__()
38          self.setupUi(self)
39          self.board = Board()
40          self.settings = QtCore.QSettings("ComputingProjectAlex", "Chess")
41          self.from_cell = []
42          self.chess_board.horizontalHeader().setResizeMode(QtGui.QHeaderView.Stretch)
43          self.chess_board.verticalHeader().setResizeMode(QtGui.QHeaderView.Stretch)
44          self.chess_board.horizontalHeader().hide()
45          self.chess_board.verticalHeader().hide()
46          self.output_board()
47          self.chess_board.itemClicked.connect(self.table_clicked)
48          self.new_btn.clicked.connect(self.new_game)
49          self.save_btn.clicked.connect(self.save_game)
50          self.load_btn.clicked.connect(self.load_game)
51
52      def table_clicked(self):
53          """Handler for when table is clicked.
54
55          When a piece has been selected to move then its legal moves are calculated.
56          These moves are then made clickable and highlighted yellow.
57          If a user selects any of the highlighted colours then the piece is
    ↪   transferred to that position
58          and a check of the game state occurs where any changes are shown in a msg
    ↪   box.
59          Promotions, if necessary, are called in this method.
60          """
61          row = self.chess_board.currentRow()
62          column = self.chess_board.currentColumn()
63          if not self.from_cell and column != -1 and row != -1:  # Piece is selected
64              self.from_cell = [row, column]
65              self.output_board(self.board.calculate_legal_moves(self.board.board[row][
                 ↪   column]))
66          elif self.from_cell and column != -1 and row != -1:  # Piece is moved
67              if self.board.board[row][column]:
68                  if self.board.board[self.from_cell[0]][self.from_cell[1]].colour ==
                     ↪   self.board.board[row][column].colour:
69                      self.from_cell = [row, column]
70                      self.output_board(self.board.calculate_legal_moves(self.board.boa
                         ↪   rd[row][column]))
```

```python
71                      else:
72                          self.board.board =
     ↪      self.board.permanently_move_piece(self.board.board,
     ↪      self.from_cell, [row, column])
73                          if self.board.must_promote:
74                              self.board.permanently_promote_piece(self.get_promotion_piece
     ↪          (), [row,
     ↪          column])
75                          self.board.check_game_state()
76                          self.output_board()
77                          self.from_cell = []
78                          if self.board.game_over and self.board.is_stalemate:
79                              self.show_message("Game is a draw. No one wins")
80                          elif self.board.game_over:
81                              self.show_message("{} is the
     ↪          winner".format(self.board.winner))
82                          elif self.board.colour_in_check:
83                              self.show_message("{} is in
     ↪          check".format(self.board.colour_in_check))
84              else:
85                  self.board.board =
     ↪      self.board.permanently_move_piece(self.board.board,
     ↪      self.from_cell, [row, column])
86                  if self.board.must_promote:
87                      self.board.permanently_promote_piece(self.get_promotion_piece(),
     ↪          [row, column])
88                  self.board.check_game_state()
89                  self.output_board()
90                  self.from_cell = []
91                  if self.board.game_over and self.board.is_stalemate:
92                      self.show_message("Game is a draw. No one wins")
93                  elif self.board.game_over:
94                      self.show_message("{} is the winner".format(self.board.winner))
95                  elif self.board.colour_in_check:
96                      self.show_message("{} is in
     ↪          check".format(self.board.colour_in_check))
97
98      def output_board(self, legal_moves=[]):
99          """Output the board onto the GUI
100
101          Uses the Board.board (2D list) to help populate the QTableWidget table.
102          All pieces of the moving player are made clickable on the table.
103          In addition, if there are legal moves specified they are all made clickable.
104
105          Args:
106              legal_moves (list): list of the legal moves a piece can make.
107          """
108          self.chess_board.clear()
109          self.chess_board.setRowCount(8)
110          self.chess_board.setColumnCount(8)
111          for y in range(8):
112              for x in range(8):
113                  if self.board.board[y][x]:
114                      if self.board.turn == "Black" and self.board.board[y][x].colour
     ↪          == "Black":
115                          item = QtGui.QTableWidgetItem()
116                          item.setSizeHint(QtCore.QSize(80, 80))
117                          item.setData(QtCore.Qt.DecorationRole,
```

```
118                                        QtGui.QPixmap(":/pieces/{}".format(self.board.bo⌋
                                        ↪  ard[y][x].img_path)))
119                              if (x+y) % 2 == 0:
120                                  item.setBackground(QtGui.QBrush(QtGui.QColor(31, 177,
                                        ↪  209)))  # light
121                              else:
122                                  item.setBackground(QtGui.QBrush(QtGui.QColor(11, 129,
                                        ↪  156)))  # dark
123                              item.setFlags(QtCore.Qt.ItemIsEnabled)
124                              self.chess_board.setItem(y, x, item)
125                          elif self.board.turn == "White" and self.board.board[y][x].colour
                              ↪  == "White":
126                              item = QtGui.QTableWidgetItem()
127                              item.setSizeHint(QtCore.QSize(80, 80))
128                              item.setData(QtCore.Qt.DecorationRole,
129                                        QtGui.QPixmap(":/pieces/{}".format(self.board.bo⌋
                                        ↪  ard[y][x].img_path)))
130                              if (x+y) % 2 == 0:
131                                  item.setBackground(QtGui.QBrush(QtGui.QColor(31, 177,
                                        ↪  209)))  # light
132                              else:
133                                  item.setBackground(QtGui.QBrush(QtGui.QColor(11, 129,
                                        ↪  156)))  # dark
134                              item.setFlags(QtCore.Qt.ItemIsEnabled)
135                              self.chess_board.setItem(y, x, item)
136                          else:
137                              item = QtGui.QTableWidgetItem()
138                              item.setSizeHint(QtCore.QSize(80, 80))
139                              item.setData(QtCore.Qt.DecorationRole,
140                                        QtGui.QPixmap(":/pieces/{}".format(self.board.bo⌋
                                        ↪  ard[y][x].img_path)))
141                              if (x+y) % 2 == 0:
142                                  item.setBackground(QtGui.QBrush(QtGui.QColor(31, 177,
                                        ↪  209)))  # light
143                              else:
144                                  item.setBackground(QtGui.QBrush(QtGui.QColor(11, 129,
                                        ↪  156)))  # dark
145                              if [y, x] in legal_moves:
146                                  item.setFlags(QtCore.Qt.ItemIsEnabled)
147                                  item.setBackground(QtGui.QBrush(QtGui.QColor(227, 209,
                                        ↪  16))) # highlight legal moves
148                              else:
149                                  item.setFlags(QtCore.Qt.NoItemFlags)
150                              self.chess_board.setItem(y, x, item)
151                      elif [y, x] in legal_moves:
152                          item = QtGui.QTableWidgetItem()
153                          item.setSizeHint(QtCore.QSize(80, 80))
154                          item.setBackground(QtGui.QBrush(QtGui.QColor(227, 209, 16))) #
                              ↪  highlight legal moves
155                          item.setFlags(QtCore.Qt.ItemIsEnabled)
156                          self.chess_board.setItem(y, x, item)
157                      else:
158                          item = QtGui.QTableWidgetItem()
159                          item.setSizeHint(QtCore.QSize(80, 80))
160                          if (x+y) % 2 == 0:
161                              item.setBackground(QtGui.QBrush(QtGui.QColor(31, 177, 209)))
                                  ↪  # light
162                          else:
```

```python
163                         item.setBackground(QtGui.QBrush(QtGui.QColor(11, 129, 156)))
                            ↪   # dark
164                     item.setFlags(QtCore.Qt.NoItemFlags)
165                     self.chess_board.setItem(y, x, item)
166
167     def new_game(self):
168         """Creates a new game with initial board setup."""
169         self.board = Board()
170         self.from_cell = []
171         self.output_board()
172
173     def load_game(self):
174         """Loads a game from a JSON file. If there is no JSON file the user is
            ↪   prompted for one.
175
176         Raises:
177             IOError: raised when there is an error loading a file.
178             FileNotFoundError: raised when a file is not found.
179             TypeError: raised when there is an error loading a file.
180             KeyError: raised when there is corruption in the JSON file.
181         """
182         try:
183             data = None
184             with open(self.settings.value('json_location')) as json_file:
185                 data = json.load(json_file)
186             game_loader = LoadDialogController(data)
187             game_loader.exec()
188             game = []
189             if game_loader.chosen_game_id != 0:
190                 for temp_game in data['games']:
191                     if game_loader.chosen_game_id == temp_game['id']:
192                         game = temp_game
193                         break
194                 self.board = Board(game)
195                 self.player_one_edit.setText(game['player_one'])
196                 self.player_two_edit.setText(game['player_two'])
197                 self.board.check_game_state()
198                 self.output_board()
199                 if self.board.game_over and self.board.is_stalemate:
200                     self.show_message("Game is a draw. No one wins")
201                 elif self.board.game_over:
202                     self.show_message("{} is the winner".format(self.board.winner))
203                 elif self.board.colour_in_check:
204                     self.show_message("{} is in
                        ↪   check".format(self.board.colour_in_check))
205         except (IOError, FileNotFoundError, TypeError):
206             self.show_message("Game file not found!")
207             self.get_load_path()
208             if self.settings.value:
209                 self.show_message("Press Load game again.")
210         except (KeyError):
211             self.show_message("Data file is corrupt. Please choose another file")
212             self.get_load_path()
213             if self.settings.value:
214                 self.load_game("Press Load game again.")
215
216     def save_game(self):
```

```
217         """Saves a currently played game, either in an existing JSON file or a new
       ↪    one.

218

219         Raises:
220             IOError: raised when there is an error loading a file.
221             FileNotFoundError: raised when a file is not found.
222         """
223         if self.player_one_edit.text() != "" and self.player_two_edit.text() != "":
224             self.board.player_one = self.player_one_edit.text()
225             self.board.player_two = self.player_two_edit.text()
226             self.board.check_game_state()
227             if self.board.winner == "White":
228                 self.board.winner = self.player_one_edit.text()
229             elif self.board.winner == "Black":
230                 self.board.winner = self.player_one_edit.text()
231             now = datetime.datetime.now()
232             year = str(now.year)
233             month = int(now.month)
234             day = int(now.day)
235             if month < 10:
236                 month = "0" + str(month)
237             else:
238                 month = str(month)
239             if day < 10:
240                 day = "0" + str(day)
241             else:
242                 day = str(day)
243             game = {
244                 'id': self.board.id,
245                 'last_played': str(year + "/" + month + "/" + day),
246                 'turn': self.board.turn,
247                 'move_num': self.board.move_num,
248                 'winner': self.board.winner,
249                 'colour_in_check': self.board.colour_in_check,
250                 'is_stalemate': self.board.is_stalemate,
251                 'game_over': self.board.game_over,
252                 'must_promote': self.board.must_promote,
253                 'enpassant_possible': {
254                     'Black': self.board.enpassant_possible['Black'],
255                     'White': self.board.enpassant_possible['White']
256                     },
257                 'enpassant_move': {
258                     'from': self.board.enpassant_move['from'],
259                     'to': self.board.enpassant_move['to'],
260                     'taken': self.board.enpassant_move['taken']
261                     },
262                 'player_one': self.board.player_one,
263                 'player_two': self.board.player_two,
264                 'pieces': {
265                     'kings': [],
266                     'queens': [],
267                     'knights': [],
268                     'bishops': [],
269                     'rooks': [],
270                     'pawns': []
271                 }
272             }
273             for row in self.board.board:
```

```
274                    for field in row:
275                        if field:
276                            if isinstance(field, Queen):
277                                piece = {
278                                    'position': field.position,
279                                    'colour': field.colour
280                                 }
281                                game['pieces']['queens'].append(piece)
282                            elif isinstance(field, Bishop):
283                                piece = {
284                                    'position': field.position,
285                                    'colour': field.colour
286                                }
287                                game['pieces']['bishops'].append(piece)
288                            elif isinstance(field, Knight):
289                                piece = {
290                                    'position': field.position,
291                                    'colour': field.colour
292                                }
293                                game['pieces']['knights'].append(piece)
294                            elif isinstance(field, Rook):
295                                piece = {
296                                    'position': field.position,
297                                    'colour': field.colour,
298                                    'has_moved': field.has_moved
299                                }
300                                game['pieces']['rooks'].append(piece)
301                            elif isinstance(field, King):
302                                piece = {
303                                    'position': field.position,
304                                    'colour': field.colour,
305                                    'has_moved': field.has_moved,
306                                    'castling_moves': field.castling_moves
307                                }
308                                game['pieces']['kings'].append(piece)
309                            elif isinstance(field, Pawn):
310                                piece = {
311                                    'position': field.position,
312                                    'colour': field.colour,
313                                    'first_moved': field.first_moved
314                                }
315                                game['pieces']['pawns'].append(piece)
316            if self.settings.value('json_location'):
317                try:
318                    data = None
319                    with open(self.settings.value('json_location')) as json_file:
320                        data = json.load(json_file)
321                    id_list = [x['id'] for x in data['games']]
322
323                    algorithms.quick_sort(id_list, 0, len(id_list)-1)
324                    if game['id']:
325                        if algorithms.binary_search(game['id'], id_list):
326                            for x in range(len(data['games'])):
327                                if game['id'] == data['games'][x]['id']:
328                                    data['games'][x] = game
329                        else:
330                            game['id'] = id_list[-1] + 1
331                            self.board.id = id_list[-1] + 1
```

```python
332                            data['games'].append(game)
333                        else:
334                            game['id'] = id_list[-1] + 1
335                            self.board.id = id_list[-1] + 1
336                            data['games'].append(game)
337                        with open(self.settings.value('json_location'), 'w') as jsonfile:
338                            json.dump(data, jsonfile, indent=4, separators=(',', ':'))
339                            self.show_message("Game saved at
     ↪   {}".format(self.settings.value('json_location')))
340                    except (IOError, FileNotFoundError):
341                        self.show_message("Error: File not found")
342                        self.settings.setValue("json_location", "")
343                        self.save_game()
344                else:
345                    self.get_save_path()
346                    if self.settings.value('json_location'):
347                        data = {'games': []}
348                        game['id'] = 1
349                        self.board.id = 1
350                        data['games'].append(game)
351                        try:
352                            with open(self.settings.value('json_location'), 'w') as
     ↪   jsonfile:
353                                json.dump(data, jsonfile, indent=4, separators=(',', ':'))
354                                self.show_message("Game saved at
     ↪   {}".format(self.settings.value('json_location')))
355                        except (FileNotFoundError, OSError):
356                            self.show_message("File not saved. Invalid file name.")
357                            self.settings.setValue("json_location", "")
358            else:
359                self.show_message("Please fill in the player names")
360
361    def get_load_path(self):
362        """Gets the JSON file to load"""
363        filepath = QtGui.QFileDialog().getOpenFileName(self, "Load Game",
     ↪   QtCore.QDir.homePath(),  "Game file (*.json)")
364        if filepath[0]:
365            self.settings.setValue("json_location", filepath[0])
366        else:
367            self.settings.setValue("json_location", "")
368            self.show_message("File not chosen.")
369
370    def get_save_path(self):
371        """Gets the desired path and filename of the file to save."""
372        filepath = QtGui.QFileDialog().getSaveFileName(self, "Save as",
     ↪   QtCore.QDir.homePath(), "Game file (*.json)")
373        if filepath[0]:
374            self.settings.setValue("json_location", filepath[0])
375        else:
376            self.settings.setValue("json_location", "")
377            self.show_message("File not saved")
378
379    def get_promotion_piece(self):
380        """Gets the piece that needs to be promoted"""
381        choice = ["", False]
382        while not choice[1]:
383            choice = QtGui.QInputDialog.getItem(self, "Piece to promote", "Choose
     ↪   piece:", ["Queen", "Knight", "Rook", "Bishop"], 0, False)
```

```python
384            return choice[0]
385
386    def show_message(self, msg):
387        """If there are errors, they are shown in a message box.
388
389        Args:
390            msg (str): message for the user to be shown.
391            """
392        msg_box = QtGui.QMessageBox()
393        msg_box.setWindowTitle("Game State")
394        msg_box.setText(msg)
395        msg_box.exec_()
396
397
398 class LoadDialogController(QtGui.QDialog, views.LoadDialog):
399    """Controller for the load dialog
400
401    When 'Load Game' is pressed the user is shown this dialog if a JSON file is found
↪     and has been loaded.
402    The user is shown a table where they can sort the games and can also choose which
↪     game to load.
403
404    Attributes:
405        chosen_game_id (int): id of the game that was chosen
406        data (dict): List of games.
407        """
408
409    def __init__(self, data):
410        """Necessary variables are initialised and table is shown to the user."""
411        super(LoadDialogController, self).__init__()
412        self.setupUi(self)
413        self.chosen_game_id = 0
414        self.data = data
415        self.output_table()
416        self.results_table.setSelectionBehavior(QtGui.QAbstractItemView.SelectRows)
417        self.results_table.setSelectionMode(QtGui.QAbstractItemView.SingleSelection)
418        self.sort_btn.clicked.connect(self.sort)
419
420    def output_table(self):
421        """List of games is shown to the user with this method."""
422        self.results_table.setRowCount(0)
423        self.buttonBox.accepted.connect(self.get_game)
424        i = 0
425        for game in self.data['games']:
426            identifier = QtGui.QTableWidgetItem()
427            identifier.setText(str(game['id']))
428            player_one = QtGui.QTableWidgetItem()
429            player_one.setText(game['player_one'])
430            player_two = QtGui.QTableWidgetItem()
431            player_two.setText(game['player_two'])
432            winner = QtGui.QTableWidgetItem()
433            winner.setText(game['winner'])
434            moves_made = QtGui.QTableWidgetItem()
435            moves_made.setText(str(game['move_num']-1))
436            last_played = QtGui.QTableWidgetItem()
437            last_played.setText(game['last_played'])
438            self.results_table.insertRow(i)
439            self.results_table.setItem(i, 0, identifier)
```

```
440                    self.results_table.setItem(i, 1, player_one)
441                    self.results_table.setItem(i, 2, player_two)
442                    self.results_table.setItem(i, 3, winner)
443                    self.results_table.setItem(i, 4, moves_made)
444                    self.results_table.setItem(i, 5, last_played)
445                    i += 1
446
447        def sort(self):
448            """Sorts the list of games"""
449            sorted_games = []
450            if self.sortby_box.currentText() == "Ascending":
451                if self.sort_type.currentText() == "ID":
452                    array = []
453                    for x in range(self.results_table.rowCount()):
454                        array.append(int(self.results_table.item(x, 0).text()))
455                    algorithms.quick_sort(array, 0, len(array)-1)
456                    sorted_games = [None] * len(array)
457                    for i in range(len(array)):
458                        for j in range(len(self.data['games'])):
459                            if array[i] == self.data['games'][j]['id']:
460                                sorted_games[i] = copy.deepcopy(self.data['games'][j])
461                                self.data['games'][j]['id'] = None
462                                break
463                elif self.sort_type.currentText() == "Player 1":
464                    array = []
465                    for x in range(self.results_table.rowCount()):
466                        array.append(self.results_table.item(x, 1).text())
467                    algorithms.quick_sort(array, 0, len(array)-1)
468                    sorted_games = [None] * len(array)
469                    for i in range(len(array)):
470                        for j in range(len(self.data['games'])):
471                            if array[i] == self.data['games'][j]['player_one']:
472                                sorted_games[i] = copy.deepcopy(self.data['games'][j])
473                                self.data['games'][j]['player_one'] = None
474                                break
475                elif self.sort_type.currentText() == "Player 2":
476                    array = []
477                    for x in range(self.results_table.rowCount()):
478                        array.append(self.results_table.item(x, 2).text())
479                    algorithms.quick_sort(array, 0, len(array)-1)
480                    sorted_games = [None] * len(array)
481                    for i in range(len(array)):
482                        for j in range(len(self.data['games'])):
483                            if array[i] == self.data['games'][j]['player_two']:
484                                sorted_games[i] = copy.deepcopy(self.data['games'][j])
485                                self.data['games'][j]['player_two'] = None
486                                break
487                elif self.sort_type.currentText() == "Winner":
488                    array = []
489                    for x in range(self.results_table.rowCount()):
490                        array.append(self.results_table.item(x, 3).text())
491                    algorithms.quick_sort(array, 0, len(array)-1)
492                    sorted_games = [None] * len(array)
493                    for i in range(len(array)):
494                        for j in range(len(self.data['games'])):
495                            if array[i] == self.data['games'][j]['winner']:
496                                sorted_games[i] = copy.deepcopy(self.data['games'][j])
497                                self.data['games'][j]['winner'] = None
```

```python
498                        break
499            elif self.sort_type.currentText() == "Moves Made":
500                array = []
501                for x in range(self.results_table.rowCount()):
502                    array.append(int(self.results_table.item(x, 4).text()))
503                algorithms.quick_sort(array, 0, len(array)-1)
504                sorted_games = [None] * len(array)
505                for i in range(len(array)):
506                    for j in range(len(self.data['games'])):
507                        if int(array[i])+1 == self.data['games'][j]['move_num']:
508                            sorted_games[i] = copy.deepcopy(self.data['games'][j])
509                            self.data['games'][j]['move_num'] = None
510                            break
511            elif self.sort_type.currentText() == "Last Played":
512                array = []
513                for x in range(self.results_table.rowCount()):
514                    array.append(self.results_table.item(x, 5).text())
515                algorithms.quick_sort(array, 0, len(array)-1)
516                sorted_games = [None] * len(array)
517                for i in range(len(array)):
518                    for j in range(len(self.data['games'])):
519                        if array[i] == self.data['games'][j]['last_played']:
520                            sorted_games[i] = copy.deepcopy(self.data['games'][j])
521                            self.data['games'][j]['last_played'] = None
522                            break
523        else:
524            if self.sort_type.currentText() == "ID":
525                array = []
526                for x in range(self.results_table.rowCount()):
527                    array.append(int(self.results_table.item(x, 0).text()))
528                algorithms.quick_sort(array, 0, len(array)-1)
529                array = array[::-1]  # reverse list
530                sorted_games = [None] * len(array)
531                for i in range(len(array)):
532                    for j in range(len(self.data['games'])):
533                        if array[i] == self.data['games'][j]['id']:
534                            sorted_games[i] = copy.deepcopy(self.data['games'][j])
535                            self.data['games'][j]['id'] = None
536                            break
537            elif self.sort_type.currentText() == "Player 1":
538                array = []
539                for x in range(self.results_table.rowCount()):
540                    array.append(self.results_table.item(x, 1).text())
541                algorithms.quick_sort(array, 0, len(array)-1)
542                array = array[::-1]  # reverse list
543                sorted_games = [None] * len(array)
544                for i in range(len(array)):
545                    for j in range(len(self.data['games'])):
546                        if array[i] == self.data['games'][j]['player_one']:
547                            sorted_games[i] = copy.deepcopy(self.data['games'][j])
548                            self.data['games'][j]['player_one'] = None
549                            break
550            elif self.sort_type.currentText() == "Player 2":
551                array = []
552                for x in range(self.results_table.rowCount()):
553                    array.append(self.results_table.item(x, 2).text())
554                algorithms.quick_sort(array, 0, len(array)-1)
555                array = array[::-1]  # reverse list
```

```python
556                     sorted_games = [None] * len(array)
557                     for i in range(len(array)):
558                         for j in range(len(self.data['games'])):
559                             if array[i] == self.data['games'][j]['player_two']:
560                                 sorted_games[i] = copy.deepcopy(self.data['games'][j])
561                                 self.data['games'][j]['player_two'] = None
562                                 break
563                 elif self.sort_type.currentText() == "Winner":
564                     array = []
565                     for x in range(self.results_table.rowCount()):
566                         array.append(self.results_table.item(x, 3).text())
567                     algorithms.quick_sort(array, 0, len(array)-1)
568                     array = array[::-1]   # reverse list
569                     sorted_games = [None] * len(array)
570                     for i in range(len(array)):
571                         for j in range(len(self.data['games'])):
572                             if array[i] == self.data['games'][j]['winner']:
573                                 sorted_games[i] = copy.deepcopy(self.data['games'][j])
574                                 self.data['games'][j]['winner'] = None
575                                 break
576                 elif self.sort_type.currentText() == "Moves Made":
577                     array = []
578                     for x in range(self.results_table.rowCount()):
579                         array.append(int(self.results_table.item(x, 4).text()))
580                     algorithms.quick_sort(array, 0, len(array)-1)
581                     array = array[::-1]   # reverse list
582                     sorted_games = [None] * len(array)
583                     for i in range(len(array)):
584                         for j in range(len(self.data['games'])):
585                             if int(array[i])+1 == self.data['games'][j]['move_num']:
586                                 sorted_games[i] = copy.deepcopy(self.data['games'][j])
587                                 self.data['games'][j]['move_num'] = None
588                                 break
589                 elif self.sort_type.currentText() == "Last Played":
590                     array = []
591                     for x in range(self.results_table.rowCount()):
592                         array.append(self.results_table.item(x, 5).text())
593                     algorithms.quick_sort(array, 0, len(array)-1)
594                     array = array[::-1]   # reverse list
595                     sorted_games = [None] * len(array)
596                     for i in range(len(array)):
597                         for j in range(len(self.data['games'])):
598                             if array[i] == self.data['games'][j]['last_played']:
599                                 sorted_games[i] = copy.deepcopy(self.data['games'][j])
600                                 self.data['games'][j]['last_played'] = None
601                                 break
602             self.data['games'] = sorted_games
603             self.output_table()
604
605         def get_game(self):
606             """When game is chosen the ID is then found."""
607             if self.results_table.currentRow() != -1:
608                 self.chosen_game_id =
                    ↪   int(self.results_table.item(self.results_table.currentRow(),
                    ↪   0).text())
609             else: # if no game selected
610                 self.chosen_game_id = 0 # Will not attempt to load game if it is 0.
```

pieces.py

```python
__author__ = "Alexander Saoutkin"


class Piece(object):
    """Base class for all chess pieces

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): Colour of the piece, either "White" or "Black"
    """

    def __init__(self, position, colour):
        """This constructor initialises the class variables and also calculates all
        ↪   possible moves for the piece."""
        self.position = position
        self.colour = colour
        self.possible_moves = []
        self.calculate_possible_moves()

        @property
        def position(self):
            return self.position

        @position.setter
        def position(self, value):
            """ This setter calculates the new possible moves once the position has
            ↪   changed.

            Args:
                value (list): coordinates on the board in the form [x, y]
            """
            self.position = value
            self.calculate_possible_moves()

    def calculate_possible_moves(self):
        """Shows us the coords it can go to assuming that there are no other pieces
        ↪   on the board

        Raises:
            NotImplementedError: Method not overridden in subclass.
        """
        raise NotImplementedError


class Rook(Piece):
    """Class for a Rook

    Attributes:
        position (list): coordinates on the board in the form [x, y]
        colour (str): colour of the piece, either "White" or "Black"
        img_path (str): path to the image of the piece
        has_moved (bool): Denotes whether the piece has moved before.
    """

    def __init__(self, position, colour, has_moved=False):
        """
```

```
54            This constructor initialises the class variables and also calculates all
  ↪    possible moves for the piece.
55            In addition the image path is added.
56            """
57            super().__init__(position, colour)
58            self.img_path = "{}_rook.png".format(self.colour.lower())
59            self.has_moved = has_moved
60
61        def calculate_possible_moves(self):
62            """Calculates all the possible moves for a rook in a certain position."""
63            self.possible_moves = []
64            for i in range(8):
65                if self.position[1] != i:
66                    self.possible_moves.append([self.position[0], i])
67                if self.position[0] != i:
68                    self.possible_moves.append([i, self.position[1]])
69
70
71    class Knight(Piece):
72        """Class for a Knight
73
74        Attributes:
75            position (list): coordinates on the board in the form [x, y]
76            colour (str): colour of the piece, either "White" or "Black"
77            img_path (str): path to the image of the piece
78        """
79
80        def __init__(self, position, colour):
81            """
82            This constructor initialises the class variables and also calculates all
  ↪    possible moves for the piece.
83            In addition the image path is added as an attribute self.img_path.
84            """
85            super().__init__(position, colour)
86            self.img_path = "{}_knight.png".format(self.colour.lower())
87
88        def calculate_possible_moves(self):
89            """Calculates all the possible moves for a knight in a certain position."""
90            self.possible_moves = []
91            potential_moves = [[self.position[0] + 2, self.position[1] + 1],
92                               [self.position[0] + 2, self.position[1] - 1],
93                               [self.position[0] - 2, self.position[1] + 1],
94                               [self.position[0] - 2, self.position[1] - 1],
95                               [self.position[0] + 1, self.position[1] + 2],
96                               [self.position[0] + 1, self.position[1] - 2],
97                               [self.position[0] - 1, self.position[1] + 2],
98                               [self.position[0] - 1, self.position[1] - 2]]
99            for move in potential_moves:
100               if 7 >= move[0] >= 0 and 7 >= move[1] >= 0:
101                   self.possible_moves.append(move)
102
103
104   class Bishop(Piece):
105       """Class for a Bishop
106
107       Attributes:
108           position (list): coordinates on the board in the form [x, y]
109           colour (str): colour of the piece, either "White" or "Black"
```

```python
110              img_path (str): path to the image of the piece
111          """
112      def __init__(self, position, colour):
113          """
114          This constructor initialises the class variables and also calculates all
     ↪   possible moves for the piece.
115          In addition the image path is added as an attribute self.img_path.
116          """
117          super().__init__(position, colour)
118          self.img_path = "{}_bishop.png".format(self.colour.lower())
119
120      def calculate_possible_moves(self):
121          """Calculates all the possible moves for a bishop in a certain position."""
122          self.possible_moves = []
123          on_edge = False
124          counter = 0
125          # Dont' bother iterating to the right if already on the edge
126          if self.position[0] is 7 or self.position[1] is 7:
127              on_edge = True
128          # Find all positions diagonally  to south-east.
129          while not on_edge:
130              counter += 1
131              self.possible_moves.append([self.position[0]+counter,
                 ↪   self.position[1]+counter])
132              if self.position[0]+counter is 7 or self.position[1]+counter is 7:
133                  on_edge = True
134          counter = 0
135          on_edge = False
136          # Dont' bother iterating to the right if already on the edge
137          if self.position[0] is 0 or self.position[1] is 0:
138              on_edge = True
139          # Find all positions diagonally to north-west
140          while not on_edge:
141              counter += 1
142              self.possible_moves.append([self.position[0]-counter,
                 ↪   self.position[1]-counter])
143              if self.position[0]-counter is 0 or self.position[1]-counter is 0:
144                  on_edge = True
145          counter = 0
146          on_edge = False
147          # Find all positions diagonally to north-east
148          if self.position[0] is 0 or self.position[1] is 7:
149              on_edge = True
150          while not on_edge:
151              counter +=1
152              self.possible_moves.append([self.position[0]-counter,
                 ↪   self.position[1]+counter])
153              if self.position[0]-counter is 0 or self.position[1]+counter is 7:
154                  on_edge = True
155          counter = 0
156          on_edge = False
157          # Find all positions diagonally to south-west
158          if self.position[0] is 7 or self.position[1] is 0:
159              on_edge = True
160          while not on_edge:
161              counter +=1
162              self.possible_moves.append([self.position[0]+counter,
                 ↪   self.position[1]-counter])
```

```python
163                     if self.position[0]+counter is 7 or self.position[1]-counter is 0:
164                         on_edge = True
165
166
167 class Queen(Piece):
168     """Class for a Queen
169
170     Attributes:
171         position (list): coordinates on the board in the form [x, y]
172         colour (str): colour of the piece, either "White" or "Black"
173         img_path (str): path to the image of the piece
174     """
175     def __init__(self, position, colour):
176         """
177         This constructor initialises the class variables and also calculates all
    ↪   possible moves for the piece.
178         In addition the image path is added as an attribute self.img_path.
179         """
180         super().__init__(position, colour)
181         self.img_path = "{}_queen.png".format(self.colour.lower())
182
183     def calculate_possible_moves(self):
184         """Calculates all the possible moves for a queen in a certain position."""
185         self.possible_moves = []
186         on_edge = False
187         counter = 0
188         # Do not bother iterating if already on the edge
189         if self.position[0] is 7 or self.position[1] is 7:
190             on_edge = True
191         # Find all positions diagonally  to south-east.
192         while not on_edge:
193             counter += 1
194             self.possible_moves.append([self.position[0]+counter,
                ↪   self.position[1]+counter])
195             if self.position[0]+counter is 7 or self.position[1]+counter is 7:
196                 on_edge = True
197         counter = 0
198         on_edge = False
199         if self.position[0] is 0 or self.position[1] is 0:
200             on_edge = True
201         # Find all positions diagonally to north-west
202         while not on_edge:
203             counter += 1
204             self.possible_moves.append([self.position[0]-counter,
                ↪   self.position[1]-counter])
205             if self.position[0]-counter is 0 or self.position[1]-counter is 0:
206                 on_edge = True
207         counter = 0
208         on_edge = False
209         # Find all positions diagonally to north-east
210         if self.position[0] is 0 or self.position[1] is 7:
211             on_edge = True
212         while not on_edge:
213             counter +=1
214             self.possible_moves.append([self.position[0]-counter,
                ↪   self.position[1]+counter])
215             if self.position[0]-counter is 0 or self.position[1]+counter is 7:
216                 on_edge = True
```

```python
217             counter = 0
218             on_edge = False
219             # Find all positions diagonally to south-west
220             if self.position[0] is 7 or self.position[1] is 0:
221                 on_edge = True
222             while not on_edge:
223                 counter +=1
224                 self.possible_moves.append([self.position[0]+counter,
                      ↪    self.position[1]-counter])
225                 if self.position[0]+counter is 7 or self.position[1]-counter is 0:
226                     on_edge = True
227             # Find all positions horizontally and vertically
228             for i in range(8):
229                 if self.position[1] != i:
230                     self.possible_moves.append([self.position[0], i])
231                 if self.position[0] != i:
232                     self.possible_moves.append([i, self.position[1]])


235 class King(Piece):
236     """Class for a King
237
238     Attributes:
239         position (list): coordinates on the board in the form [x, y].
240         colour (str): colour of the piece, either "White" or "Black".
241         img_path (str): path to the image of the piece.
242         has_moved (bool): denotes whether piece has moved or not.
243         castling_moves (list): list of all castling moves possible.
244     """
245     def __init__(self, position, colour, has_moved=False, castling_moves=[]):
246         """
247         This constructor initialises the class variables and also calculates all
    ↪   possible moves for the piece.
248         In addition the image path is added as an attribute self.img_path.
249         """
250         super().__init__(position, colour)
251         self.img_path = "{}_king.png".format(self.colour.lower())
252         self.has_moved = has_moved
253         self.castling_moves = castling_moves
254
255     def calculate_possible_moves(self):
256         """Calculates all the possible moves for a king in a certain position."""
257         self.possible_moves.clear()
258         potential_moves = []
259         potential_moves.append([self.position[0]+1, self.position[1]+1])
260         potential_moves.append([self.position[0]+1, self.position[1]-1])
261         potential_moves.append([self.position[0]+1, self.position[1]])
262         potential_moves.append([self.position[0]-1, self.position[1]+1])
263         potential_moves.append([self.position[0]-1, self.position[1]-1])
264         potential_moves.append([self.position[0]-1, self.position[1]])
265         potential_moves.append([self.position[0], self.position[1]+1])
266         potential_moves.append([self.position[0], self.position[1]-1])
267         for move in potential_moves:
268             if move[0] <= 7 and move [0] >= 0 and move[1] <= 7 and move[1] >= 0:
269                 self.possible_moves.append(move)


272 class Pawn(Piece):
```

```
273     """Class for a Pawn
274
275     Attributes:
276         position (list): coordinates on the board in the form [x, y]
277         colour (str): colour of the piece, either "White" or "Black"
278         img_path (str): path to the image of the piece
279         first_moved (int): denotes at what stage (move) in the game the the piece was
    ↪  first moved.
280     """
281     def __init__(self, position, colour, first_moved=0):
282         """
283         This constructor initialises the class variables and also calculates all
    ↪  possible moves for the piece.
284         In addition the image path is added as an attribute self.img_path.
285         """
286         super().__init__(position, colour)
287         self.img_path = "{}_pawn.png".format(self.colour.lower())
288         self.first_moved = first_moved
289
290     def calculate_possible_moves(self):
291         """Calculates all the possible moves for a pawn in a certain position."""
292         self.possible_moves.clear()
293         if self.colour == "White":
294             self.possible_moves.append([self.position[0]-1, self.position[1]])
295         else:
296             self.possible_moves.append([self.position[0]+1, self.position[1]])
```

views.py

```
1   from PySide import QtCore, QtGui
2   import resources_rc
3
4   class MainWindow(object):
5       def setupUi(self, MainWindow):
6           MainWindow.setObjectName("MainWindow")
7           MainWindow.resize(660, 742)
8           self.centralwidget = QtGui.QWidget(MainWindow)
9           self.centralwidget.setObjectName("centralwidget")
10          MainWindow.setCentralWidget(self.centralwidget)
11          self.menubar = QtGui.QMenuBar(MainWindow)
12          self.menubar.setGeometry(QtCore.QRect(0, 0, 660, 21))
13          self.menubar.setObjectName("menubar")
14          MainWindow.setMenuBar(self.menubar)
15          self.statusbar = QtGui.QStatusBar(MainWindow)
16          self.statusbar.setObjectName("statusbar")
17          MainWindow.setStatusBar(self.statusbar)
18
19          self.retranslateUi(MainWindow)
20          QtCore.QMetaObject.connectSlotsByName(MainWindow)
21
22      def retranslateUi(self, MainWindow):
23          MainWindow.setWindowTitle(QtGui.QApplication.translate("Chess for students",
    ↪  "Chess for students", None, QtGui.QApplication.UnicodeUTF8))
24
25  class ChessBoard(object):
26      def setupUi(self, Form):
```

```
27        Form.setObjectName("Form")
28        Form.resize(661, 728)
29        self.horizontalLayoutWidget = QtGui.QWidget(Form)
30        self.horizontalLayoutWidget.setGeometry(QtCore.QRect(9, 10, 641, 31))
31        self.horizontalLayoutWidget.setObjectName("horizontalLayoutWidget")
32        self.horizontalLayout = QtGui.QHBoxLayout(self.horizontalLayoutWidget)
33        self.horizontalLayout.setContentsMargins(0, 0, 0, 0)
34        self.horizontalLayout.setObjectName("horizontalLayout")
35        self.player_one_label = QtGui.QLabel(self.horizontalLayoutWidget)
36        self.player_one_label.setObjectName("player_one_label")
37        self.horizontalLayout.addWidget(self.player_one_label)
38        self.player_one_edit = QtGui.QLineEdit(self.horizontalLayoutWidget)
39        self.player_one_edit.setObjectName("player_one_edit")
40        self.horizontalLayout.addWidget(self.player_one_edit)
41        spacerItem = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
   ↪   QtGui.QSizePolicy.Minimum)
42        self.horizontalLayout.addItem(spacerItem)
43        self.player_two_label = QtGui.QLabel(self.horizontalLayoutWidget)
44        self.player_two_label.setObjectName("player_two_label")
45        self.horizontalLayout.addWidget(self.player_two_label)
46        self.player_two_edit = QtGui.QLineEdit(self.horizontalLayoutWidget)
47        self.player_two_edit.setObjectName("player_two_edit")
48        self.horizontalLayout.addWidget(self.player_two_edit)
49        self.horizontalLayoutWidget_2 = QtGui.QWidget(Form)
50        self.horizontalLayoutWidget_2.setGeometry(QtCore.QRect(10, 690, 641, 41))
51        self.horizontalLayoutWidget_2.setObjectName("horizontalLayoutWidget_2")
52        self.horizontalLayout_2 = QtGui.QHBoxLayout(self.horizontalLayoutWidget_2)
53        self.horizontalLayout_2.setContentsMargins(0, 0, 0, 0)
54        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
55        self.new_btn = QtGui.QPushButton(self.horizontalLayoutWidget_2)
56        self.new_btn.setObjectName("new_btn")
57        self.horizontalLayout_2.addWidget(self.new_btn)
58        self.load_btn = QtGui.QPushButton(self.horizontalLayoutWidget_2)
59        self.load_btn.setObjectName("load_btn")
60        self.horizontalLayout_2.addWidget(self.load_btn)
61        self.save_btn = QtGui.QPushButton(self.horizontalLayoutWidget_2)
62        self.save_btn.setObjectName("save_btn")
63        self.horizontalLayout_2.addWidget(self.save_btn)
64        self.chess_board = QtGui.QTableWidget(Form)
65        self.chess_board.setGeometry(QtCore.QRect(10, 50, 640, 640))
66        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Fixed,
   ↪   QtGui.QSizePolicy.Fixed)
67        sizePolicy.setHorizontalStretch(0)
68        sizePolicy.setVerticalStretch(0)
69        sizePolicy.setHeightForWidth(self.chess_board.sizePolicy().hasHeightForWidth(
   ↪   ))
70        self.chess_board.setSizePolicy(sizePolicy)
71        self.chess_board.setAutoFillBackground(False)
72        self.chess_board.setLineWidth(1)
73        self.chess_board.setEditTriggers(QtGui.QAbstractItemView.NoEditTriggers)
74        self.chess_board.setTabKeyNavigation(False)
75        self.chess_board.setProperty("showDropIndicator", False)
76        self.chess_board.setDragDropOverwriteMode(False)
77        self.chess_board.setAlternatingRowColors(False)
78        self.chess_board.setSelectionMode(QtGui.QAbstractItemView.SingleSelection)
79        self.chess_board.setSelectionBehavior(QtGui.QAbstractItemView.SelectItems)
80        self.chess_board.setTextElideMode(QtCore.Qt.ElideMiddle)
81        self.chess_board.setVerticalScrollMode(QtGui.QAbstractItemView.ScrollPerItem)
```

```
82          self.chess_board.setGridStyle(QtCore.Qt.NoPen)
83          self.chess_board.setCornerButtonEnabled(True)
84          self.chess_board.setObjectName("chess_board")
85          self.chess_board.setColumnCount(8)
86          self.chess_board.setRowCount(8)
87          item = QtGui.QTableWidgetItem()
88          self.chess_board.setVerticalHeaderItem(0, item)
89          item = QtGui.QTableWidgetItem()
90          self.chess_board.setVerticalHeaderItem(1, item)
91          item = QtGui.QTableWidgetItem()
92          self.chess_board.setVerticalHeaderItem(2, item)
93          item = QtGui.QTableWidgetItem()
94          self.chess_board.setVerticalHeaderItem(3, item)
95          item = QtGui.QTableWidgetItem()
96          self.chess_board.setVerticalHeaderItem(4, item)
97          item = QtGui.QTableWidgetItem()
98          self.chess_board.setVerticalHeaderItem(5, item)
99          item = QtGui.QTableWidgetItem()
100         self.chess_board.setVerticalHeaderItem(6, item)
101         item = QtGui.QTableWidgetItem()
102         self.chess_board.setVerticalHeaderItem(7, item)
103         item = QtGui.QTableWidgetItem()
104         self.chess_board.setHorizontalHeaderItem(0, item)
105         item = QtGui.QTableWidgetItem()
106         self.chess_board.setHorizontalHeaderItem(1, item)
107         item = QtGui.QTableWidgetItem()
108         self.chess_board.setHorizontalHeaderItem(2, item)
109         item = QtGui.QTableWidgetItem()
110         self.chess_board.setHorizontalHeaderItem(3, item)
111         item = QtGui.QTableWidgetItem()
112         self.chess_board.setHorizontalHeaderItem(4, item)
113         item = QtGui.QTableWidgetItem()
114         self.chess_board.setHorizontalHeaderItem(5, item)
115         item = QtGui.QTableWidgetItem()
116         self.chess_board.setHorizontalHeaderItem(6, item)
117         item = QtGui.QTableWidgetItem()
118         self.chess_board.setHorizontalHeaderItem(7, item)
119         self.chess_board.horizontalHeader().setVisible(False)
120         self.chess_board.horizontalHeader().setStretchLastSection(False)
121         self.chess_board.verticalHeader().setVisible(True)
122         self.chess_board.verticalHeader().setDefaultSectionSize(30)
123         self.chess_board.verticalHeader().setHighlightSections(False)
124
125         self.retranslateUi(Form)
126         QtCore.QMetaObject.connectSlotsByName(Form)
127
128     def retranslateUi(self, Form):
129         Form.setWindowTitle(QtGui.QApplication.translate("Form", "Form", None,
            ↪   QtGui.QApplication.UnicodeUTF8))
130         self.player_one_label.setText(QtGui.QApplication.translate("Form", "Player 1
            ↪   (White):", None, QtGui.QApplication.UnicodeUTF8))
131         self.player_two_label.setText(QtGui.QApplication.translate("Form", "Player 2
            ↪   (Black):", None, QtGui.QApplication.UnicodeUTF8))
132         self.new_btn.setText(QtGui.QApplication.translate("Form", "New Game", None,
            ↪   QtGui.QApplication.UnicodeUTF8))
133         self.load_btn.setText(QtGui.QApplication.translate("Form", "Load Game", None,
            ↪   QtGui.QApplication.UnicodeUTF8))
```

```
134         self.save_btn.setText(QtGui.QApplication.translate("Form", "Save Game", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
135         self.chess_board.verticalHeaderItem(0).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
136         self.chess_board.verticalHeaderItem(1).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
137         self.chess_board.verticalHeaderItem(2).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
138         self.chess_board.verticalHeaderItem(3).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
139         self.chess_board.verticalHeaderItem(4).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
140         self.chess_board.verticalHeaderItem(5).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
141         self.chess_board.verticalHeaderItem(6).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
142         self.chess_board.verticalHeaderItem(7).setText(QtGui.QApplication.translate("
        ↪    Form", "New Row", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
143         self.chess_board.horizontalHeaderItem(0).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
144         self.chess_board.horizontalHeaderItem(1).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
145         self.chess_board.horizontalHeaderItem(2).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
146         self.chess_board.horizontalHeaderItem(3).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
147         self.chess_board.horizontalHeaderItem(4).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
148         self.chess_board.horizontalHeaderItem(5).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
149         self.chess_board.horizontalHeaderItem(6).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
150         self.chess_board.horizontalHeaderItem(7).setText(QtGui.QApplication.translate
        ↪    ("Form", "New Column", None,
        ↪    QtGui.QApplication.UnicodeUTF8))
151
152 class LoadDialog(object):
153     def setupUi(self, Dialog):
154         Dialog.setObjectName("Dialog")
155         Dialog.resize(616, 454)
156         self.verticalLayoutWidget = QtGui.QWidget(Dialog)
157         self.verticalLayoutWidget.setGeometry(QtCore.QRect(0, 0, 611, 451))
158         self.verticalLayoutWidget.setObjectName("verticalLayoutWidget")
```

```
159        self.verticalLayout = QtGui.QVBoxLayout(self.verticalLayoutWidget)
160        self.verticalLayout.setSizeConstraint(QtGui.QLayout.SetNoConstraint)
161        self.verticalLayout.setContentsMargins(-1, -1, -1, 4)
162        self.verticalLayout.setObjectName("verticalLayout")
163        self.horizontalLayout = QtGui.QHBoxLayout()
164        self.horizontalLayout.setContentsMargins(0, 0, 0, 0)
165        self.horizontalLayout.setObjectName("horizontalLayout")
166        self.label = QtGui.QLabel(self.verticalLayoutWidget)
167        self.label.setObjectName("label")
168        self.horizontalLayout.addWidget(self.label)
169        self.sort_type = QtGui.QComboBox(self.verticalLayoutWidget)
170        self.sort_type.setObjectName("sort_type")
171        self.sort_type.addItem("")
172        self.sort_type.addItem("")
173        self.sort_type.addItem("")
174        self.sort_type.addItem("")
175        self.sort_type.addItem("")
176        self.sort_type.addItem("")
177        self.horizontalLayout.addWidget(self.sort_type)
178        self.sortby_box = QtGui.QComboBox(self.verticalLayoutWidget)
179        self.sortby_box.setObjectName("sortby_box")
180        self.sortby_box.addItem("")
181        self.sortby_box.addItem("")
182        self.horizontalLayout.addWidget(self.sortby_box)
183        self.sort_btn = QtGui.QPushButton(self.verticalLayoutWidget)
184        self.sort_btn.setObjectName("sort_btn")
185        self.horizontalLayout.addWidget(self.sort_btn)
186        self.verticalLayout.addLayout(self.horizontalLayout)
187        self.results_table = QtGui.QTableWidget(self.verticalLayoutWidget)
188        self.results_table.setEnabled(True)
189        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Expanding,
     ↪   QtGui.QSizePolicy.Expanding)
190        sizePolicy.setHorizontalStretch(0)
191        sizePolicy.setVerticalStretch(0)
192        sizePolicy.setHeightForWidth(self.results_table.sizePolicy().hasHeightForWidt⌋
     ↪   h())
193        self.results_table.setSizePolicy(sizePolicy)
194        self.results_table.setAutoFillBackground(False)
195        self.results_table.setFrameShadow(QtGui.QFrame.Sunken)
196        self.results_table.setEditTriggers(QtGui.QAbstractItemView.NoEditTriggers)
197        self.results_table.setWordWrap(True)
198        self.results_table.setRowCount(0)
199        self.results_table.setObjectName("results_table")
200        self.results_table.setColumnCount(6)
201        self.results_table.setRowCount(0)
202        item = QtGui.QTableWidgetItem()
203        self.results_table.setHorizontalHeaderItem(0, item)
204        item = QtGui.QTableWidgetItem()
205        self.results_table.setHorizontalHeaderItem(1, item)
206        item = QtGui.QTableWidgetItem()
207        self.results_table.setHorizontalHeaderItem(2, item)
208        item = QtGui.QTableWidgetItem()
209        self.results_table.setHorizontalHeaderItem(3, item)
210        item = QtGui.QTableWidgetItem()
211        self.results_table.setHorizontalHeaderItem(4, item)
212        item = QtGui.QTableWidgetItem()
213        self.results_table.setHorizontalHeaderItem(5, item)
214        self.verticalLayout.addWidget(self.results_table)
```

```python
215        self.buttonBox = QtGui.QDialogButtonBox(self.verticalLayoutWidget)
216        self.buttonBox.setOrientation(QtCore.Qt.Horizontal)
217        self.buttonBox.setStandardButtons(QtGui.QDialogButtonBox.Cancel |
       ↪   QtGui.QDialogButtonBox.Open)
218        self.buttonBox.setObjectName("buttonBox")
219        self.verticalLayout.addWidget(self.buttonBox)
220
221        self.retranslateUi(Dialog)
222        QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL("accepted()"),
       ↪   Dialog.accept)
223        QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL("rejected()"),
       ↪   Dialog.reject)
224        QtCore.QMetaObject.connectSlotsByName(Dialog)
225
226    def retranslateUi(self, Dialog):
227        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None,
       ↪   QtGui.QApplication.UnicodeUTF8))
228        self.label.setText(QtGui.QApplication.translate("Dialog", "Sort By:", None,
       ↪   QtGui.QApplication.UnicodeUTF8))
229        self.sort_type.setItemText(0,
230                            QtGui.QApplication.translate("Dialog", "ID", None,
                               ↪   QtGui.QApplication.UnicodeUTF8))
231        self.sort_type.setItemText(1, QtGui.QApplication.translate("Dialog", "Player
       ↪   1", None,
232                                               QtGui.QApplication⌋
                                               ↪   .UnicodeUTF8))
233        self.sort_type.setItemText(2, QtGui.QApplication.translate("Dialog", "Player
       ↪   2", None,
234                                               QtGui.QApplication⌋
                                               ↪   .UnicodeUTF8))
235        self.sort_type.setItemText(3, QtGui.QApplication.translate("Dialog",
       ↪   "Winner", None,
236                                               QtGui.QApplication⌋
                                               ↪   .UnicodeUTF8))
237        self.sort_type.setItemText(4, QtGui.QApplication.translate("Dialog", "Moves
       ↪   Made", None,
238                                               QtGui.QApplication⌋
                                               ↪   .UnicodeUTF8))
239        self.sort_type.setItemText(5, QtGui.QApplication.translate("Dialog", "Last
       ↪   Played", None,
240                                               QtGui.QApplication⌋
                                               ↪   .UnicodeUTF8))
241        self.sortby_box.setItemText(0, QtGui.QApplication.translate("Dialog",
       ↪   "Ascending", None,
242                                               QtGui.QApplicatio⌋
                                               ↪   n.UnicodeUTF⌋
                                               ↪   8))
243        self.sortby_box.setItemText(1, QtGui.QApplication.translate("Dialog",
       ↪   "Descending", None,
244                                               QtGui.QApplicatio⌋
                                               ↪   n.UnicodeUTF⌋
                                               ↪   8))
245        self.sort_btn.setText(QtGui.QApplication.translate("Dialog", "Sort", None,
       ↪   QtGui.QApplication.UnicodeUTF8))
246        self.results_table.setSortingEnabled(False)
247        self.results_table.horizontalHeaderItem(0).setText(
248            QtGui.QApplication.translate("Dialog", "ID", None,
               ↪   QtGui.QApplication.UnicodeUTF8))
```

```
249         self.results_table.horizontalHeaderItem(1).setText(
250             QtGui.QApplication.translate("Dialog", "Player 1", None,
                ↪   QtGui.QApplication.UnicodeUTF8))
251         self.results_table.horizontalHeaderItem(2).setText(
252             QtGui.QApplication.translate("Dialog", "Player 2", None,
                ↪   QtGui.QApplication.UnicodeUTF8))
253         self.results_table.horizontalHeaderItem(3).setText(
254             QtGui.QApplication.translate("Dialog", "Winner", None,
                ↪   QtGui.QApplication.UnicodeUTF8))
255         self.results_table.horizontalHeaderItem(4).setText(
256             QtGui.QApplication.translate("Dialog", "Moves Made", None,
                ↪   QtGui.QApplication.UnicodeUTF8))
257         self.results_table.horizontalHeaderItem(5).setText(
258             QtGui.QApplication.translate("Dialog", "Last Played", None,
                ↪   QtGui.QApplication.UnicodeUTF8))
```