



Building a Chess AI



Table of Contents

Analysis	4
Proposed Project	4
Overview of the Project	4
User(s)/Client	4
Project Research	5
Project Requirements	13
Must	13
Should	13
Could	13
Won't	14
Proposed Solution	14
Documented Design	14
Overall System Design	14
Inputs, Processes, Storage and Outputs	14
Design of Project Modules	17
AI Package	17
Database Package	17
Input Package	17
Entities Package	17
Screens Package	18
Utility Package	18
Game Class	18
Definition of Record Structure	19
The Move Stack	19
The Game Message Queue	19
Validating Input	19
Database Design	20
Data Dictionaries	20
Sample of Planned SQL Queries	22
Program Algorithms	23
The MiniMax Algorithm	23
The MiniMax Algorithm with Alpha-Beta	24
The Evaluation Function	24

Class Definitions and Diagrams	25
A Design of the User Interface	28
Technical Solution	29
Introduction	29
com.dylanwalsh.chessai.....	29
com.dylanwalsh.chessai.ai	29
com.dylanwalsh.chessai.database.....	34
com.dylanwalsh.chessai.entities	39
com.dylanwalsh.chessai.input.....	51
com.dylanwalsh.chessai.screens.....	53
com.dylanwalsh.chessai.util.....	67
GameClass.....	70
Assets.....	71
chess_pieces.....	71
Hud	71
tiles	71
System Testing	72
Test Tables	72
Piece Functionality Tests.....	72
Other Tests	75
Screenshots.....	80
Screenshot 1.1	80
Screenshot 1.2.....	80
Screenshot 1.3.....	80
Screenshot 1.4.....	81
Screenshot 1.5.....	81
Screenshot 1.6.....	81
Screenshot 1.7.....	82
Screenshot 1.8.....	82
Screenshot 1.9.....	83
Screenshot 2.0.....	83
Screenshot 2.1.....	84
Screenshot 2.2.....	84
Screenshot 2.3.....	85

Screenshot 2.4.....	85
Screenshot 2.7.....	85
Screenshot 2.8.....	85
Screenshot 2.9.....	86
Screenshot 3.0.....	87
Screenshot 3.1.....	87
Screenshot 3.2.....	87
Screenshot 3.3.....	88
Screenshot 3.4.....	88
Screenshot 3.5.....	88
Screenshot 3.6.....	89
Screenshot 3.7.....	89
Screenshot 3.8.....	90
Screenshot 3.9.....	90
Screenshot 4.0.....	90
Screenshot 4.1.....	90
Screenshot 4.2.....	90
Screenshot 4.3.....	90
Minimax Stats.....	91
2.5.....	91
2.6.....	93
Evaluation.....	95
Overview.....	95
Feedback.....	97
Ease of use.....	97
Improvements.....	98

Analysis

Proposed Project

Overview of the Project

Artificial Intelligence is a rapidly growing field. With a broad range of applications such as autonomous road vehicles, Google's search algorithms or IBM's Watson, AI can come in many forms. Generally, however, the different forms of AI can be broken down into two categories; narrow/weak AI and general/strong AI. What we have today is what is properly known as narrow or weak AI. This approach to AI research and development involves designing AI systems that simply act upon and are bound to the rules imposed on it. This specialized type of AI is designed to perform very specific tasks. An example of a weak AI could be video game characters who will only act according to the rules defined for them by the programmer. Some other examples of weak AI could include Apple's Siri, a chess bot or an autonomous system designed to drive a car. A much more advanced form of AI called general or strong AI is a system designed to take a more general approach to solving problems. Because of this, a general AI will be able to perform nearly any cognitive task as opposed to a specific task. The field of AGI (Artificial General Intelligence) is aimed at building general-purpose AI systems that can 'Think' and possess intelligence comparable to that of the human mind, ultimately capable of experiencing consciousness. Because designing and building a system like this is so complex, artificial general intelligence remains a primary goal of some artificial intelligence research and a common topic for science fiction and future studies.

With this in mind, I have taken on the challenge of building a chess artificial intelligence that can compete with a player/user. The chess AI will be able to determine the best possible next move for any given board state and my designed algorithm for doing this will make use of the Minimax algorithm. I hope to include three separate difficulties (easy, medium and hard) where each difficulty comes progressively more difficult to beat. I intend to eventually include database compatibility so that I could permanently store game information such as the sequence of moves made or the best score for each difficulty.

User(s)/Client

My client is a friend of mine who has tasked me with creating a chess artificial intelligence. He will be the user and owner of the software. After asking him a set of questions (detailed in the User Requirements section) I have identified the various essential features of the project and what exactly the client wants from the project.

Project Research

Having never taken on a project like this before, there is a lot of information that I don't know and designing the solution may be difficult without first researching into the problem. Here I intend to gain a better understanding of what information I will need to figure out a solution. This will include learning about potential algorithms I could use such as the MiniMax algorithm or other existing solutions to my problem (similar systems) and how they work.

Research Table

What you need to know	Source of information (Who?)	Primary Secondary	Research Method
User Requirements	Client/User	Primary	Interview – An interview will be the most appropriate method to attain the user requirements from one person as it will allow me to respond to any answers in a more specific way, which may lead me on to further questions that I didn't think about before.
Similar Systems Analysis		Primary	Investigation – An investigation into various other similar systems (other chess games, chess engines) and what I like and dislike about each.
Research of appropriate data structures		Secondary	Website – I will conduct some research into the various data structures that can be used throughout the project and where they would be appropriate to use. For example, search trees, Queues, Stacks, etc.
Research of AI algorithms, the MiniMax algorithm and Alpha-beta pruning			Website – I will research into the various AI algorithms that can be used in the game of chess. These include MiniMax, search tree

			algorithms and an enhancement on the MiniMax called Alpha-beta pruning.
Research any game languages, APIs I could use.		Secondary	Website – A Comparison of various languages and an evaluation of the most appropriate for my situation. Research into which APIs I could use with Java such as JDBC.
Research appropriate RDBMS to use in my project.		Secondary	Website – Conduct research into the various RDBMS available and evaluate which one is most suitable and why.

User Requirements

The interview I conducted with my client/user allowed me to get a better understanding of the user requirements and what the client wanted from the software. Being in the form of an interview, I could respond to each answer differently. For instance when the client told me he wanted game information stored in the database, I was able to ask him what type of data he wanted saved.

The questions I have designed for my interview to identify my user requirements for the system are as follows (I have detailed the answers also):

What features must be implemented in the final system?

A: I would like there to be a chessboard rendered on screen that the user can interact with, I would also like the algorithm used to be the MiniMax algorithm using Alpha-Beta pruning. There must also be information about the game stored in the database.

What type of information about the game would you like stored in the database?

A: I would like the move history of a game to be stored in the database and if so, the game can be played back to the user. The database could also store the player's highscore.

What other features could be implemented in the final system?

A: Extra features could include highlighted tiles when hovered over or selected, there could also be a very simple main menu system. Not all chess rules need to be implemented.

Which chess rules therefore aren't essential?

Castling and pawn en-passant.

What other features won't be implemented in the final system?

A: The chessboard shouldn't be rendered in 3D using 3D models.

What would the user interface look like?



A: The chessboard and all pieces should be rendered on screen and if so, information about the captured pieces and elapsed time since the start of the game as well as the current score of the player.




How will the user interact with the program?

A: A user should select a tile containing a piece and then select a tile to move that piece to. The user shouldn't drag and drop the chess piece.

Similar Systems Analysis

In order to better understand the problem I am trying to solve, I thought I would conduct some analysis of other chess games and see which features would be appropriate for my project, which features wouldn't and which features I won't be able to implement within reasonable time. To do this, I researched various popular chess game engines online.

System	Hardware	Software	Algorithms	User Interface
The Chessmaster series	MS Windows, MS-DOS, Game Boy, PlayStation, Xbox 360, PlayStation Portable, PlayStation 2	-	The Chessmaster chess engine is called The King written by Johan de Koning.	 <p>A screenshot of a game being played in Chessmaster Grandmaster (above) and a screenshot of the menu screen of Chessmaster Grandmaster (below). I like the simplistic design of this UI. In addition to the chessboard, I would also like to have a game HUD showing elapsed time since the start of the game.</p> 

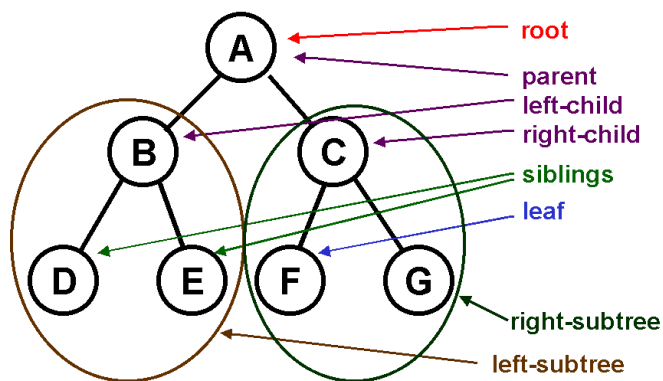
Fritz	Windows Vista, Windows XP, PlayStation 3, Wii, Nintendo DS, Windows 7, Windows 10	-	Since Fritz 15, the games have been using Vasik Rajlich's Rybka engine.	 <p>This is the UI of a Fritz 15 chess game. The 2D representation of the chess board and chess pieces in this instance are much closer to how I will render the chess board in my game. However, I do not like how the menu items, settings and game statistics take up the majority of the screen.</p>
Pure Chess	Windows, PlayStation 3, PlayStation 4, PlayStation Vita, Nintendo 3DS, Wii U, IOS, Android, Xbox One	-	-	 <p>With much more photorealistic 3D rendered graphics, Pure Chess is geared more towards producing better game visuals with more natural gameplay mechanics than Fritz and ChessMaster. Though I won't be able to render the chessboard in this way, I do like the way the HUD is laid out on screen with the transparent menu items, it isn't as clustered as a Fritz 15 game.</p> 

Data Structures

I will be using a variety of data structures throughout this project. I intend to compare a few common data structure and list appropriate uses for each, and where they would be useful in my project. In my project, there will be a lot of data passed around, and representing that data and organising it into the right structure will be essential. For instance, recording piece history would involve the use of a stack when the item at the top of the stack is the most recently moved piece.

Tree

A tree is an Abstract Data Type (ADT). It is a form of a non-linear data structure, meaning it doesn't have a definitive start and end like linear data structures do (Arrays, Queues, Linked Lists, etc.). In a tree, data is stored in a hierarchy structure. It is made up of a collection of entities called nodes which are connected via edges. Each node will contain a data item and may or may not have a child node. The top most node of a tree is called the root node and has no parents. It becomes a parent if it is connected to a sub-node. The sub-node would therefore be the child of the root node. Leaves are the last nodes on a tree, they are nodes without children. A final important concept is height and depth. The height of a node is the number of edges along the longest path between that node and a leaf. The height of the tree is the height of the root node. The depth of a node is the sum of edges from that node to the root node.



A useful implementation of a tree within my chess project could be in conjunction with the MiniMax algorithm. The search tree will represent all possible moves to a given depth. The position to move will be evaluated at the ending leaves of the tree.

Array

An array is an example of a linear data structure. It consists of a collection of elements (data values) each individually accessed by an index. An array can be either one-dimensional or multidimensional. A multidimensional array is an array containing one or more nested arrays. This can be useful when wanting to represent a collection of data in the form of a grid. In my situation I could use a two dimensional array to represent the chessboard. The array would contain 8 other nested arrays representing the 8 rows of the board. The nested arrays will contain 8 objects representing the state of each tile on that row (E.G. Pawn, Bishop, None).

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	BR	Bk	BB	BQ	BK	BB	Bk	BR
[1]	BP	BP	BP	BP	BP	BP	BP	BP
[2]								
[3]								

[4]								
[5]								
[6]	WP	WP	WP	WP	WP	WP	WP	WP
[7]	WR	Wk	WB	WQ	WK	WB	Wk	WR

Queue

A queue is a form of linear structured abstract data type. It follows a particular order in which operations should be performed. This order is known as **First In First Out (FIFO)**. An analogy for a queue data structure would be a queue of customers at a supermarket till. The first person to enter the queue (the first data item) would be the first to leave the queue. This means the item to remove from the queue would be the least recently added item.

There are four main operations that can be performed on a queue data structure:

Enqueue: Adds an item to the end of the queue, in the case that a queue is full, it is known as an overflow condition.

Dequeue: Removes an item from the front of the queue, in the case that a queue is empty, it is known as an underflow condition.

Front: Retrieve the front item of the queue

Rear: Retrieve the last item of the queue

Stack

A stack is another form of linear abstract data structure and is similar to a queue. Much like a stack or pile in real life, items are removed and added to a stack from the top. Because of this, a stack is known as a **Last In First Out (LIFO)** structure. There are 3 main operations that you can perform on a stack:

Push: Insert an item to the top of the stack

Pop: Remove an item from the top of the stack

Pip: Display the contents of the stack

The MiniMax Algorithm

The most important aspect of my project will be move generation, which means having the AI determine the best possible move given any state of the board. However, the problem with this is that the number of states that a chessboard can be in is far too large to store. This means you cannot pre-determine the relative value of any state of the board, rather you will have to measure this value through the use of an evaluation function. This, in conjunction with the minimax algorithm is what will allow the AI to figure out the best move to make.

The MiniMax algorithm is an algorithm used in decision making and game theory to find the most optimal move for a player. It is mostly effective in two player turn based games such as chess, tic-tac-toe, mancala, etc. In the MiniMax algorithm, the two players are called the maximizer and the minimizer. Both aiming for opposite outcomes, the maximizer attempts to obtain the highest score possible while the minimiser tries to obtain the lowest score possible.

Each state of the board will have a unique value associated with it. In any given state, if the maximizer has the upper hand the board score will tend to take on some positive value. Whereas if the minimizer has the upper hand, it will be some negative value. Every type of game will have some unique heuristics, which are used to calculate particular values of the board.

Alpha-Beta pruning

With larger search depths for the minimax algorithm, there will be more nodes on the search tree to examine. This can leave the program using more resources and compute time. Alpha-beta pruning is an enhancement on the minimax algorithm.

The MiniMax using a search tree could potentially end up evaluating a very large number of nodes. Alpha-beta pruning is a search algorithm that can help solve this problem by decreasing the number of nodes that are evaluated. It will stop evaluating a move when just one possibility has been found that proves that the move will leave a piece in a worse position than a previously evaluated move. Because this node, and subsequent child nodes in the tree, are no longer evaluated, it means there are less comparisons and leaves the overall algorithm much faster and more efficient.

Languages

There are a range of programming languages available to solve my problem. From various programming paradigms to the different APIs and frameworks available to each language, I will examine the pros and cons of a few potential languages that I could use.

Java

I could use the Java programming language alongside the game framework LibGDX. This would be helpful as Java is a language I am most familiar with and have used LibGDX on multiple projects before. With this experience, I won't need to spend time learning how to use a particular library or new language which will give me more time to develop my project. I would also use the JDBC API for the database connectivity. This would be helpful as the JDBC API comes as part of the Java SDK within the `java.sql` and `javax.sql` packages. This is much more convenient as I won't need to install any extra libraries.

Python

If I were to use Python, I would have to learn a new game framework or set of libraries (such as Pygame and DB-API) for the rendering of graphics on screen or dealing with the database connectivity. While I do have experience working with Python, learning a new library could take too long. Python is very user friendly and generally an easier language to learn, however python is much more of a functional language than the more object oriented Java. I think an object oriented language would be most appropriate for my project as it allows me to split the complexity into multiple classes and group similar processes together. An object oriented language will offer a higher level of abstraction than a language such as Python. When creating a complex project, this

allows me to take a more top down approach to programming and will leave my project more maintainable and easier to understand.

C#

C# is a language I am least experienced with. I have used it before but would need to spend time learning its syntax and libraries more. If I were to use C# I could even use a game engine such as Unity. A game engine would be helpful as it will do a lot of the heavy lifting for me. This would leave me focusing more on the content and layout of the game. There are other 2D game frameworks for C# such as FlatRedBall. While C# is a multi-paradigm language, it is very object oriented like Java.

JavaScript

I have used JavaScript before and created games using the Canvas API, since it isn't a compiled language and can run in the browser, it means development time can be much quicker and wouldn't require anything more than a text editor and web browser. There are also various open source HTML5 game frameworks to choose from like Phaser. However, JavaScript is also more of a functional language which does not support classes, and since my project will benefit from being object oriented, this language will probably not be my best option.

Final choice

I have chosen to use Java with the LibGDX game framework and the JDBC API. I believe an object oriented language would be most appropriate for my project. Java is the language I have the most experience with and having used LibGDX and JDBC before, I will be able to focus more on figuring out how the game will function rather than learning the syntax of the language or how to use the library. Both Java and LibGDX are also very well documented.

LibGDX

LibGDX is an open source game development framework written in the Java programming language. It uses a Gradle based build system and allows for cross platform game development where your project can be deployed to Windows, Linux, Mac OS X, Android, Blackberry, iOS and HTML5 using WebGL. LibGDX has various APIs available for common game development tasks such as rendering graphics, building UIs, playing back audio, parsing XML and JSON, etc.

Using a game framework like this for my project would mean I would have to worry less about how the low level processes work, leaving me focusing more on how to create the project itself. It also cuts down on development time, with code more concise and easy to understand.

Relational Database Management Systems

Similar to how there are different languages that I could use, there are also a selection of RDBMS that are available. Choosing the right RDBMS is an important aspect of my project as updates to the database will be automatic after every game and pulling data from the database will need to be a quick process. This means I will need to consider both speed and capacity when choosing my RDBMS.

MySQL

MySQL is the most widely used of all RDBMS and easy to work with. It is feature rich but not too complex either. MySQL is also very fast when dealing with the more simple queries such as primary key lookups, ranged queries, etc. MySQL also still maintains good performance when storing much

larger amounts of data, however this is only for the more simple queries and can struggle to keep up with the more complicated queries.

PostgreSQL

One of the biggest advantages of PostgreSQL is that it handles well with larger amounts of data and more complex data models. It, like MySQL, is also free and open source. PostgreSQL also has support for lots of libraries and frameworks. It is also used often in conjunction with Python. However, installation and configuration can be complicated at times and isn't the most beginner friendly of RDBMS. It also has a very large feature set which means that it can take a long time to learn to use properly.

SQLite

SQLite is a very lightweight RDBMS and good for embedded software. It also has good performance, is easy to learn and installation and configuration isn't complicated. SQLite is generally more suited to smaller scale projects. SQLite however cannot handle larger amounts of HTTP requests/traffic. This means it is not suited to handling larger scale projects.

Final choice

I have chosen to use MySQL for my project as it can handle all operations I intend to perform and can maintain good performance with larger datasets. It is also easy to configure and manage for a project like mine.

Project Requirements

Must

When the program is first run, it must render a chess board on screen along with all the chess pieces in the correct place.

The program must allow the user to move their chess pieces on the board according to the rules of chess.

The program must calculate a next move based upon the current state of the board using the MiniMax algorithm and alpha-beta pruning.

Should

There should be functionality for loading previously played games to rewatch, or continue playing if the game isn't finished. This data should be store in a database and updated after every game.

There should be a load game button on screen to load any previous game.

The rule of castling should be implemented with both the user and AI performing the move.

Could

The only rule which isn't essential is pawn en-passant. This move doesn't occur very often within a game of chess and so implementing it won't be a necessity.

Tiles could change colour when selected or hovered over.

I could implement a main menu system (This would be in the form of a separate screen).

The game could display data on the side of the screen about time elapsed for the player, how many moves have been made, taken pieces and a message box to prompt the user on the move made by the AI, whether the king is in check or there is a checkmate, which piece a pawn was just promoted to, etc.

Won't

I won't be rendering out a 3D view of the chess board or create any models of the board and pieces. It will be a top down 2D view of the board.

Proposed Solution

I will be writing this project in the Java programming language using the LibGDX game framework. With LibGDX, loading and managing assets, rendering graphics, dealing with user input and managing the game camera and viewport for different window sizes will be much easier. I will be using JDBC to interact with the database, allowing me to store information about the game. To make shore I meet all user requirements, my solution will have to meet the following criteria:

- The game board will have to be rendered on screen with all chess pieces in the correct place.
- The movement of each chess piece will have to abide by the official rules of chess.
- The user interacts with the board by using a mouse.
- The algorithm used to calculate the best move will be the MiniMax algorithm using alpha-beta pruning.
- Information stored in the database will include move history and the player high score (number of moves taken to check mate the AI).
- There will be functionality to play back a previously saved game.
- Along with the board, information on the time elapsed since the start of the game and captured pieces will be displayed on screen.
- The game will allow both the player and AI to perform a castling move.
- The game will allow both the player and AI to perform a pawn en-passant move.

Documented Design

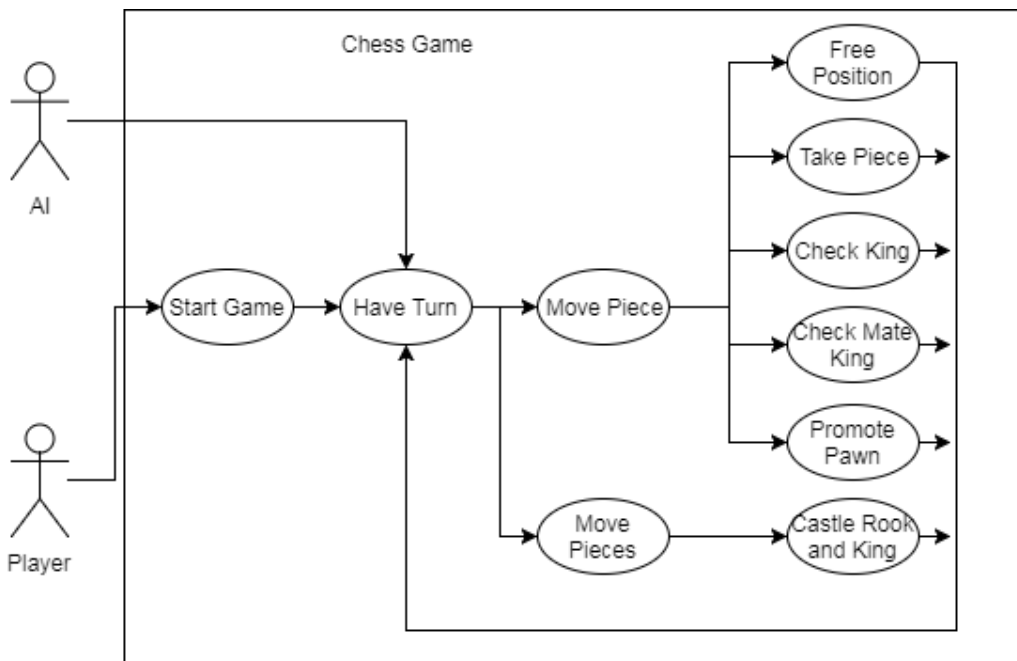
Overall System Design

Inputs, Processes, Storage and Outputs

Inputs	Processes	Storage	Outputs
Selecting and moving a white chess piece to empty space.	Update new board positions, record move and generate new moves for all pieces. Increment move counter. If this resulted in a finished game, input to the board is disabled.	Move is recorded in a move stack to later be placed in the database.	Board is rendered with piece moved to new position. Display time elapsed for player turn, the move counter and a prompt telling the user which piece the AI moved and to what position, whether this resulted in the king being in check or if there is a checkmate.

<p>Selecting and moving a white chess piece to take a black piece.</p>	<p>Update new board positions, remove and dispose taken piece, record move and generate new moves for all pieces. Increment move counter. If this resulted in a finished game, input to the board is disabled.</p>	<p>Move is recorded in a move stack to later be placed in the database.</p>	<p>Board is rendered with pieces moved to new position. The taken piece is removed from the board and placed in a taken pieces table on screen to the side of the board. Display time elapsed for player turn, the move counter and a prompt telling the user which piece the AI moved and to what position, whether this resulted in the king being in check or if there is a checkmate.</p>
<p>Enter username in the text field.</p>	<p>No process until the record game button is pressed</p>	<p>No storage until the record game button is pressed</p>	<p>Display the username in the text field.</p>
<p>Record game button is pressed.</p>	<p>If this is a brand-new game, and if there is no username in the text field, the user is prompted to enter a username. If this is an old game being resaved, then the text field would have been disabled and there will be no username in the text field. The game so far is then saved to the database depending on whether the game is being updated or saved and the user is added to the database if it is a new user. The board is disabled if the game is complete. However, the game won't be saved to the database if it is an old game that has already been completed. The load game button is automatically pressed</p>	<p>Game information is stored in the database. In the users table this includes the user id, the name and high-score. In the games table this includes game id, a time stamp, the time elapsed for the player, the score/moves made, the move history, the user id and the win loss state.</p>	<p>A prompt to the user is displayed on screen if no username is entered for a new game save. 'Game Saved' or 'Game Updated' is displayed on screen depending on whether this save was a game update or new game save. All games in the database are display on screen in a scrollable table. The format for displaying a game is: Name : time-stamp : score : win/loss/incomplete</p>

	to display the games and the new saved game. A stored procedure in the database is then called to update the high_score value in the users table.		
Load game button is pressed.	All games are retrieved from the database and displayed on screen in a scrollable table.	Nothing is saved when loading games from the database.	All games in the database are display on screen in a scrollable table. The format for displaying a game is: Name : time-stamp : score : win/loss/incomplete
A game is the game table is clicked to play.	The board is updated to the state of the game at the last save. If this was a finished game, input to the board is disabled.	Nothing is stored when a game is loaded.	The time elapsed for the game, the number of moves, the game prompts and pieces taken for that game are displayed on screen. The board is rendered to the loaded game state.



Design of Project Modules

Details of each package/module will be described fully in the technical solution. Here I intend to give an overview of what each I intend each package's purpose is and what files and stored in each package.

AI Package

This package will encapsulate everything to do with the AI calculating its best move for a given state of the board. There will be 3 files in this package, 2 or which are used with the evaluation function (function used to evaluate how advantageous or disadvantageous a state of the board is).

Class: GameAI

This will contain the search algorithm and evaluation function for the game AI.

Class: PieceSquareTables

This will be a static class containing all the piece square tables for each piece on the board. A piece square table is just a simple 8x8 table containing values for each tile of the board that a piece is either better off (some positive value) or not better off (some negative value) in.

Class: RelativePieceValues

This will also be a static class containing the relative piece values for each piece. For instance, a queen would be 900, a rook would be 500 and a pawn would be 100. This would mean 1 rook and 4 pawns are equivalent to a queen.

Database Package

This package will contain all code needed to interact with the database and store information about a game. There are 2 files in the package.

Class: DBConnection

This will be a static class. Its main purpose is to contain all functionality for saving a game, loading a game/games.

Class: GameData

This class will be used to represent a game/row in the database. Each instance will contain unique values corresponding to the values of the row in the database (id, score, move history, etc.).

Input Package

This class will contain just 1 file that deals with the game input.

Class: GameInput

This class will keep track of any key presses, mouse movements, mouse clicks for the board. This does not deal with the input for the HUD.

Entities Package

This package will contain information about all chess pieces. All unique child chess pieces inside entities.chesspieces will inherit from the parent abstract ChessPiece class.

Class: ChessPiece

This will be an abstract class to represent a chess piece and contain everything common to each piece. This will be the parent class to all other chess pieces.

Entities.Chesspieces

This sub-package will contain all child class implementations of ChessPiece. This includes the classes:

- Bishop
- King
- Knight
- Pawn
- Queen
- Rook

Screens Package

This is where the three most important components of my project will live; the Board, GameScreen and HUD.

Class: GameScreen

This class will be composed of an instance of both the Board and HUD. It will deal with setting up all necessities for the project.

Class: Board

This class will be used to represent the board, updating and rendering the tiles and pieces in response to user input.

Class: HUD

This class will be used to represent the HUD, positioning and updating all widgets.

Utility Package

This package will contain all functionality for moving a piece. This would involve taking other pieces, abiding by the castling rules and undoing moves. This is important as the minimax algorithm (the search algorithm for the AI part of my code) works by looking a certain number of moves ahead.

Class: PieceMovements

This static class will perform and undo piece movements. It will keep track of the actual movements in a stack.

Class: PieceMovement

This class will represent a single piece movement by keeping track of information such as the piece to move, where to move it from, where to move it to, what the piece is at the position to move to if there is a piece there, etc.

Game Class

This class is the starting point of the application and is necessary for the game library I am using. It just creates an instance of GameScreen.

Definition of Record Structure

There are areas of my project that call for specific use of data structures. I will list a few examples of the structures and where I will use them

The Move Stack

When considering how to keep track of the movements that occur throughout the course of the game, I found it important to think about how some moves will be undone. This would mean that the undone movements would not become part of the final move sequence for the finished game. The best way to represent this information would therefore be a stack. Every move made will be pushed to the top of the stack and moves undone will be popped off the top. Moves that are not undone will remain on the stack as final moves made (the final move chosen by the minimax algorithm for instance) and will become part of the finished sequence of moves for the game.

The Game Message Queue

My program will need to display messages to the user throughout the game such as where a black piece moved to, whether the game was updated or saved, whether the player's king is in check, etc. Messages will be displayed in sequence and therefore more than one message could be displayed at one time. For example, a message prompting the user where the AI moved a piece to and that the player's king is in check. To properly order these messages, I will use a queue. The message queue will contain 3 items at one time, each representing a message to display on screen. The newest message added to the queue will be displayed at the bottom of the message box and older messages will be displayed above. This way the player can see which messages are the most recent.

Validating Input

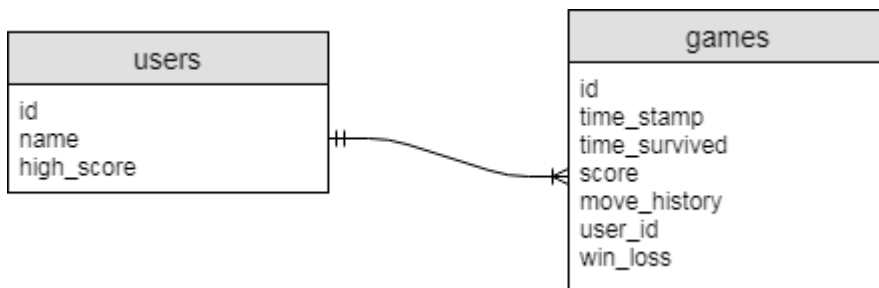
There will be two main ways that a user can enter data into the program; through the game board and the text field. The following table details how I validate each of these inputs.

Validation Check	Description	Suitable Data	Erroneous Data	Displayed Error Message/Actions Taken for Erroneous Data
Lookup/List	A check that the position a player wants to move a piece to is within the pieces move set.	A coordinate (x, y) that is within a piece's move set.	A coordinate (x, y) that is not within a piece's move set.	Board is not updated.
Lookup/List	A check that the position a player wants to move a piece to doesn't result in the king being in check	A coordinate (x, y) that is within a piece's move set that doesn't leave the king in check.	A coordinate (x, y) that is within a piece's move set that doesn't leave the king in check.	Displayed message 'King in Check!'
Presence	A check that the username text field contains data if this is a new game.	A string of length greater than 0.	No string entered.	Displayed message 'Enter a Username!'

Length	The username entered must be less than or equal to 20.	String of length less than or equal to 20.	A string greater than 20	Displayed message 'Max Username Length is 20!'
--------	--	--	--------------------------	--

Database Design

Within my database there will be two main tables: the games table and the users table. The games table will all information about all games including a user id foreign key corresponding to the id of the users table. The users table will store the name of the user as well as the high-score. The following E-R Diagram shows how I will lay out my tables as well as the relationships between them.



There is a foreign key reference in the games table to the users table; user_id. The user_id in games corresponds to a primary key of the users table. The same user_id can occur in multiple rows. There is a one-to-many relationship between users and games which means that one user can have many different games.

Data Dictionaries

Users Table

Name	Data Type	Description	Example (Valid, Invalid)
id	SMALLINT	The primary key of the users table, uniquely identifies a user.	3 -2 (though SMALLINT can store negative numbers, I do not want there to be negative ids).
name	VARCHAR(20)	The name of the user, cannot exceed 20 characters.	Dylan 123username321username
high_score	SMALLINT	The high score for this user.	12 -10 (It is impossible to have a negative high score)

Games Table

Name	Data Type	Description	Example
id	SMALLINT	The primary key of games table, uniquely identifies a game.	14 -8 (I also do not want to store negative ids for the games table).
time_stamp	DATE	The date the game was last saved.	2018-02-15 2030-04-29 (Future date), 2017-09-32 (Invalid date)
time_survived	SMALLINT	The elapsed time for the game at the last save.	259 (2:59) 4 (0:04) 63 (0:63) – This should be represented as 103 (1:03) -201 (-2:01) – Cannot have a negative time
score	SMALLINT	The number of moves for the game at the last save.	20 -10 (Cannot make a negative number of moves).
move_history	TEXT	A string representing all game moves at the last save. The format for a move would be: previousX previousY newX newY (pawnPromotion character) Where previousX, previousY, newX and newY are numbers in the range 0-7 representing indexes of the 2D board array. The pawn promotion character would only appear if a pawn was promoted as part of a move made. Some examples include... 3545 would mean the piece at 6D moved to 6E. 6667Q would mean the white pawn at G7 moved to G8 and was promoted to a Queen.	11136755... – start of game, white pawn move followed by black knight move. ...6160K... - black pawn promotion to a knight. ...7787... – outside board range ...1617L... – invalid pawn promotion character.

user_id	SMALLINT	The foreign key to the users id in the users table.	8 -3 (I do not want negative ids). Number not matching an id in users.
win_loss	TINYINT	An integer to track whether the game was a win (0), a loss (1) or an unfinished game (2) at the last save.	0, 1, 2 Anything else.

Sample of Planned SQL Queries

I will be using a range of different queries throughout my project to retrieve data and update data in the database. All these queries will be parameterized so that I can insert values into the queries much easier.

Data Definition Language

```
CREATE TABLE users(
    id SMALLINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(20) NOT NULL,
    high_score SMALLINT NOT NULL DEFAULT 0,
    PRIMARY KEY(id)
)
```

I will be using a stored procedure to update high_score, so giving this column a default value of 0 means that I won't have to provide a value in the UPDATE query in my program.

```
CREATE TABLE games (
    id SMALLINT NOT NULL AUTO_INCREMENT,
    time_stamp DATE NOT NULL,
    time_survived SMALLINT NOT NULL,
    score SMALLINT NOT NULL,
    move_history TEXT NOT NULL,
    user_id SMALLINT NOT NULL,
    win_loss BOOLEAN NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY(user_id) REFERENCES users(id)
)
```

TEXT character large object can store 64KB, a string of 65535 characters in length. With a move being represented by four separate integers, that means TEXT can store a game with 16383 moves. Which is more moves that can be made in a single game.

I will set the user_id column to be a foreign key referencing the id in the users table.

ALTER

I have decided that I need to make a change to the games table as I want to store 3 separate values for win_loss; 0 for loss, 1 for win and 2 for unfinished. So, rather than constructing the entire table again, I use the following query.

```
ALTER TABLE games MODIFY COLUMN win_loss TINYINT NOT NULL;
```

UPDATE

```
UPDATE games SET time_stamp=(SELECT CURDATE()), time_survived=?, score=?,  
move_history=?, win_loss=? WHERE id=?;
```

I will use this query to make an update to a row in the games table. This will update an already existing game in the database. For that I will need to know the id of the game and I won't be updating the user_id of the game as I will already have that stored. Rather than setting finding the date in the code and then setting that value for time_stamp, I will use a sub-query SELECT CURDATE() which returns the current date in YYYY-MM-DD format.

SELECT

```
SELECT * FROM users WHERE name=?;
```

This is a very simple query that I will use to check if a row in the database with a given name already exists. Using this, I will only save new users to the users table, which means there is no repeated data.

INSERT

```
INSERT INTO users(name) VALUES(?);
```

I will use this query to insert a new user into the users table. I do not need to provide an id or high_score as the id of the newly added user will be the next available id (AUTO_INCREMENT) and the high_score will take on a default value of 0, to later be updated by a stored procedure.

```
INSERT INTO games(time_stamp, time_survived, score, move_history, user_id,  
win_loss) VALUES((SELECT CURDATE()), ?, ?, ?, (SELECT id FROM users WHERE name =  
?), ?);
```

This is the most complex of the queries I will use in my program. This query will be used to insert a new game into the games table. The only value I do not need to pass is the id, as it will be automatically incremented with a new insert (brand-new game). I will again be using the SELECT CURDATE() sub-query to get the current date the game will be saved on. I will also use another sub-query (SELECT id FROM users WHERE name = ?), which will allow me to look up the value of the user id in the users table given that I know the name of the user. Because there won't be any users with the same username, this means there will only be one user id for any given name.

Program Algorithms

There are various algorithms I will be using in my project. The most complex of which will be the MiniMax algorithm. This is the search algorithm for the game AI. It has an improvement called alpha-beta pruning which will decrease the number of nodes evaluated, meaning it can run faster and search at deeper depths. This is used in conjunction with an evaluation function, which can consider a multitude of factors when calculating a value for the board state. The following are pseudo code representations of how I intend to implement both my search algorithm and evaluation function.

The MiniMax Algorithm

```
function MiniMax(depth, maxi)  
  if maxi is true then  
    if depth is 0 then  
      return EvaluateBoard()  
    max = -∞  
    for all moves  
      score = MiniMax(depth-1, !maxi)  
      max = MaximumOf(score, max)  
    return max  
  else  
    if depth is 0 then  
      return -EvaluateBoard()
```



```

min = +∞
for all moves
    score = MiniMax(depth-1, !maxi)
    min = MinimumOf(score, min)
return min

```

This is the pseudo code of the minimax algorithm. It will return a value when for the leaf nodes (at a depth of 0) and for nodes that aren't leaf nodes, their value is taken from a descendant leaf node. The value of a leaf node is calculated by an evaluation function which produces some heuristic value representing the favourability of a node (game state) for the maximising player. Nodes that lead to a better outcome for the maximising player will therefore take on higher values than nodes that are more favourable for the minimizing player.

The MiniMax Algorithm with Alpha-Beta

```

function MiniMaxWithAlphaBeta(depth, alpha, beta, maxi)
    if maxi is true then
        if depth is 0 then
            return EvaluateBoard()
        max = -∞
        for all moves
            score = MiniMaxWithAlphaBeta(depth-1, alpha, beta, !maxi)
            max = MaximumOf(score, max)
            alpha = MaximumOf(alpha, max)
            if beta <= alpha then
                break out of loop
        return max
    else
        if depth is 0 then
            return -EvaluateBoard()
        min = +∞
        for all moves
            score = MiniMaxWithAlphaBeta(depth-1, alpha, beta, !maxi)
            min = MinimumOf(score, min)
            beta = MinimumOf(beta, min)
            if beta <= alpha then
                break out of loop
        return min

```

The alpha beta pruning is an enhancement on the minimax algorithm that introduces two new values; alpha and beta. Alpha represents the best score for max along the path to the current state of the game. Beta represents the best score for min along the path to the current state of the game.

The Evaluation Function

The evaluation function will probably be the most important aspect of the game AI. There are various aspects to an evaluation function, some of the most important include:

- Material – The sum of piece values for each side.
- Relative Piece Values – A value assigned to each piece when calculating its relative strength in comparison to another piece.
- Mobility – A measure of how many legal moves can be made in a given state of the game.
- King Safety – An evaluation of how safe the king is in any given state of the board.
- Piece-Square tables – A way of assigning specific piece values to specific board positions.

There are other parts of an evaluation function that you can consider but for my project I want to just use material, relative piece values and piece-square tables to determine the value of a state of the board. So, my evaluation function will look like the following.

```

function EvaluateBoard()
    total = 0
    for all pieces on the board
        total += GetPieceValue(piece)
    return total

function GetPieceValue(piece)
    value = 0
    value += PieceRelativeValue(piece)
    value += PieceSquareTableValue(piece)
    if piece is white then
        return value
    else
        return -value

```

I will sum every piece's relative value and square table reference on the board and return a negative value for each black piece and a positive value for each white piece. The EvaluateBoard function will return a unique evaluation of the board at that state.

Class Definitions and Diagrams

Class	Methods and Properties (Method : Return Type) / (Property : Data Type)
GameAI	<pre> calculateBestMove(ChessPiece[][]): void miniMaxAlphaBeta(int, int, int, boolean): int evaluateBoard(): int getPieceValue(ChessPiece): int getPieceSquareTableValue(ChessPiece): int getPieceRelativeValue(ChessPiece): int getPieces(boolean): ArrayList<ChessPiece> getPiece(): ChessPiece getMove(): Vector2 searchDepth: int = 3 board: ChessPiece[][] piece: ChessPiece move: Vector2 </pre>
PieceSquareTables	<pre> reverseTable(short[][]): short[][] PAWNTABLE: short[][] KNIGHTTABLE: short[][] BISHOPTABLE: short[][] ROOKTABLE: short[][] QUEENTABLE: short[][] KINGTABLE: short[][] </pre>

RelativePieceValues	PAWN: int = 100 KNIGHT: int = 320 BISHOP: int = 330 ROOK: int = 500 QUEEN: int = 900 KING: int = 20000
DBConnection	saveGame(String, int, int, String, int, int): GameData getLatestGame(int): GameData getPastGames(): ArrayList<GameData>
GameData	GameData(int, Date, int, int, String, String, int) getGameId(): int getTimeStamp(): Date getTimeSurvived(): int getScore(): int getMoveHistory(): String getName(): String getWin(): int
Bishop	Bishop(Pieces, int, int) generatePositions(ChessPiece[][]): void
King	King(Pieces, int, int, Rook, Rook) isInCheck(ChessPiece[][]): boolean setCastlingPositions(int, int, int, int): void canCastleLeft(): boolean canCastleRight(): boolean getCastleLeftRook(): Rook getCastleRightRook(): Rook getCastleLeftPosition(): Vector2 getCastleRightPosition(): Vector2 setCanCastleLeft(boolean): void setCanCastleRight(boolean): void setMoved(): void hasMoved(): boolean generatePositions(ChessPiece[][]): void
Knight	Knight(Pieces, int, int) generatePositions(ChessPiece[][]): void
Pawn	Pawn(Pieces, int, int) generatePositions(ChessPiece[][]): void promote(ChessPiece[], char): char
Queen	Queen(Pieces, int, int) generatePositions(ChessPiece[][]): void
Rook	Rook(Pieces, int, int, char) setCastlingPosition(int, int): void getType(): char setMoved(): void hasMoved(): boolean getCastlePosition(): Vector2 generatePositions(ChessPiece[][]): void

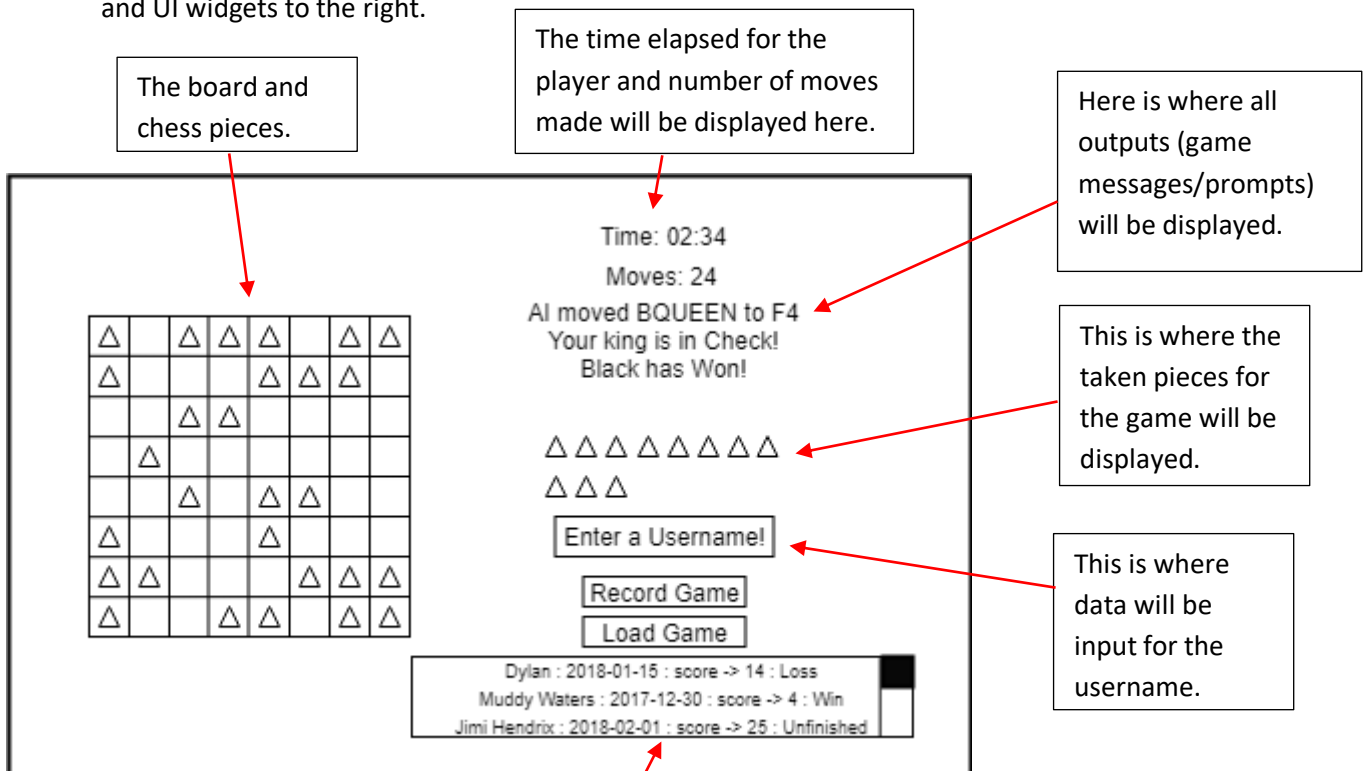
ChessPiece	<pre> ChessPiece(Pieces, int, int) getPieceTexture(): Texture moveTo(int, int, ChessPiece[][]): boolean getFriendlyKing(ChessPiece[][]): King isCheckingKing(): boolean getAllPotentialMoves(): ArrayList<Vector2> generatePositions(ChessPiece[][]): void getPiece(): Pieces getPosition(): Vector2 setPosition(Vector2): void dispose(): void </pre>	
GameInput	<pre> GameInput(int, int, Vector2, ArrayList<Vector2>, OrthographicCamera, ChessPiece[][], HUD) keyDown(int): boolean keyUp(int): boolean promotePawn(Pawn, char): char keyTyped(char): boolean touchDown(int, int, int, int): boolean touchUp(int, int, int, int): boolean touchDragged(int, int, int): boolean mouseMoved(int, int): boolean scrolled(int): boolean setGameOver(): void </pre>	
Board	<pre> Board(SpriteBatch, OrthographicCamera, HUD) setGame(String): void renderBoard(): boolean getInput(): GameInput dispose(): void </pre>	
GameScreen	<pre> GameScreen() setGame(String): void show(): void render(float): void resize(int, int): void pause(): void resume(): void hide(): void dispose(): void VIEWPORT_WIDTH: float = 370 </pre>	
HUD	<pre> HUD(GameScreen) changeTurn(): void setGameOver(int): void setGameMoves(String): void addTakenPiece(ChessPiece): void addGameMessage(String): void renderHUD(): void getInput(): Stage dispose(): void </pre>	

PieceMovement	<pre> PieceMovement(ChessPiece, int, int, ChessPiece, boolean) isCastle(): boolean getPiece(): ChessPiece getPieceAtPosition(): ChessPiece getFromX(): int getFromY(): int getToX(): int getToY(): int </pre>
PieceMovements	<pre> move(ChessPiece[][], ChessPiece, int, int): ChessPiece undo(): void getGameMoves(): String resetMoves(): void recordLastMove(): void promoteWhitePawn(Pawn, char, GameInput): void promoteBlackPawn(Pawn, char): void </pre>
GameClass	<pre> create(): void render(): void dispose(): void </pre>

See file named 'ChessUML.jpeg' for the UML class digram.

A Design of the User Interface

I intend to keep my user interface simple and easy to understand. With this in mind, I am going to keep everything on one screen; with the chessboard and rendered pieces to the left and the HUD and UI widgets to the right.



Technical Solution

Introduction

com.dylanwalsh.chessai

com.dylanwalsh.chessai.ai

GameAI

```
package com.dylanwalsh.chessai.ai;

import com.badlogic.gdx.math.Vector2;
import com.dylanwalsh.chessai.entities.ChessPiece;
import com.dylanwalsh.chessai.util.PieceMovements;

import java.util.ArrayList;

/**
 * This class deals with all functionality concerned with generating the best move
 * for the AI. It does this by using the minimax
 * algorithm with the alpha-beta pruning enhancement.
 */
public class GameAI {
    //The depth to explore the search tree.
    private int searchDepth = 3;
    //A copy of the board.
    private ChessPiece[][] board;
    //The piece and move evaluated by AI for the current turn.
    private ChessPiece piece;
    private Vector2 move;

    /**
     * This method is called to assign values to piece and move.
     * @param board the 2D board array.
     */
    public void calculateBestMove(ChessPiece[][] board) {
        this.board = board.clone();
        piece = null;
        move = null;
        miniMaxAlphaBeta(searchDepth, -11111, 11111, false);
    }

    /**
     * The minimax search algorithm, which traverses the search tree of potential
     * future moves and evaluates the best move for any
     * given game state (state of the board) down to a specific depth. This is a
     * recursive algorithm which calls itself.
     * @param depth the depth to explore the search tree.
     * @param alpha alpha value for improved minimax algorithm.
     * @param beta beta value for improved minimax algorithm.
     * @param maxi true if this is the maximizing player, false if this is the
     * minimizer.
     * @return a score/value measuring the favorability of a particular node at a
     * particular depth.
     */
    private int miniMaxAlphaBeta(int depth, int alpha, int beta, boolean maxi) {
        if(maxi) {
```

```

        if(depth == 0) return evaluateBoard();
        int max = -9999;
        for(ChessPiece p : getPieces(maxi)) for(Vector2 m :
p.getAllPotentialMoves()) {
            PieceMovements.move(board, p, (int)m.x, (int)m.y);
            int score = miniMaxAlphaBeta(depth-1, alpha, beta, !maxi);
            try {
                if(p.getFriendlyKing(board).isInCheck(board)) score = -9999;
            } catch(NullPointerException e) {
                score = -9999;
            }
            PieceMovements.undo();
            if(score > max) {
                max = score;
                if(depth == searchDepth) {
                    piece = p;
                    move = m;
                }
            }
            alpha = Math.max(alpha, max);
            if(beta<=alpha) return max;
        }
        return max;
    } else {
        if(depth == 0) return -evaluateBoard();
        int min = 9999;
        for(ChessPiece p : getPieces(maxi)) for(Vector2 m :
p.getAllPotentialMoves()) {
            PieceMovements.move(board, p, (int)m.x, (int)m.y);
            int score = miniMaxAlphaBeta(depth-1, alpha, beta, !maxi);
            try {
                if(p.getFriendlyKing(board).isInCheck(board)) score = 9999;
            } catch(NullPointerException e) {
                score = 9999;
            }
            PieceMovements.undo();
            if(score < min) {
                min = score;
                if(depth == searchDepth) {
                    piece = p;
                    move = m;
                }
            }
            beta = Math.min(beta, min);
            if(beta<=alpha) return min;
        }
        return min;
    }
}

/**
 * The method used to evaluate the state of the board, which considers all
 * pieces on the board. It uses a pieces relative piece
 * value as well as looking up it piece square table to determine its value.
 *
 * @return an evaluation/integer value of the board.
 */
private int evaluateBoard() {
    int total = 0;
    for(ChessPiece[] row : board) {
        for(ChessPiece p : row) {
            if(p!=null) {
                total += getPieceValue(p);
            }
        }
    }
    return total;
}
}

```

```

/**
 * This method retrieves a given pieces value on the board. It uses the piece's
 * relative piece value and looks up it's piece
 * square table value to determine a value for the piece.
 * @param p the ChessPiece to evaluate.
 * @return the chess piece's value on the board
 */
private int getPieceValue(ChessPiece p) {
    int value = 0;
    value += getPieceRelativeValue(p);
    value += getPieceSquareTableValue(p);
    return (p.getPiece().toString().charAt(0)=='W'?value:-value);
}

/**
 * This method looks up a piece's position on it's piece square table.
 * @param p the piece to look up
 * @return a value at the piece's current position on it's piece square table
 * representing how favorable that position is for that piece.
 */
private int getPieceSquareTableValue(ChessPiece p) {
    short[][] table = null;
    switch(p.getPiece()) {
        case WPAWN:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.PAWN_TABLE);
            break;
        case BPAWN:
            table = PieceSquareTables.PAWN_TABLE;
            break;
        case WKNIGHT:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.KNIGHT_TABLE);
            break;
        case BKNIGHT:
            table = PieceSquareTables.KNIGHT_TABLE;
            break;
        case WROOK:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.ROOK_TABLE);
            break;
        case BROOK:
            table = PieceSquareTables.ROOK_TABLE;
            break;
        case WBISHOP:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.BISHOP_TABLE);
            break;
        case BBISHOP:
            table = PieceSquareTables.BISHOP_TABLE;
            break;
        case WQUEEN:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.QUEEN_TABLE);
            break;
        case BQUEEN:
            table = PieceSquareTables.QUEEN_TABLE;
            break;
        case WKING:
            table =
PieceSquareTables.reverseTable(PieceSquareTables.KING_TABLE);
            break;
        case BKING:
            table = PieceSquareTables.KING_TABLE;
            break;
    }
    return table[(int)p.getPosition().y][(int)p.getPosition().x];
}

```



```

}

/**
 * Looks up this pieces relative value.
 * @param p the piece to look up.
 * @return the piece's relative value.
 */
private int getPieceRelativeValue(ChessPiece p) {
    switch(p.getPiece()) {
        case WPAWN:
        case BPAWN:
            return RelativePieceValues.PAWN;
        case WKNIGHT:
        case BKNIGHT:
            return RelativePieceValues.KNIGHT;
        case WROOK:
        case BROOK:
            return RelativePieceValues.ROOK;
        case WBISHOP:
        case BBISHOP:
            return RelativePieceValues.BISHOP;
        case WQUEEN:
        case BQUEEN:
            return RelativePieceValues.QUEEN;
        case WKING:
        case BKING:
            return RelativePieceValues.KING;
    }
    return 0;
}

/**
 * @param isWhite if true, returns all white pieces, if false returns all black
pieces.
 * @return a list of all white or black pieces on the board.
 */
private ArrayList<ChessPiece> getPieces(boolean isWhite) {
    ArrayList<ChessPiece> pieces = new ArrayList<ChessPiece>();
    for(ChessPiece[] row : board)
        for(ChessPiece p : row)
            if(p != null) {
                if(isWhite) {
                    if(p.getPiece().toString().charAt(0) == 'W') pieces.add(p);
                } else {
                    if(p.getPiece().toString().charAt(0) == 'B') pieces.add(p);
                }
            }
    return pieces;
}

/**
 * @return the next piece to move determined by the minimax algorithm.
 */
public ChessPiece getPiece() { return piece; }

/**
 * @return the position to move the piece to determined by the minimax
algorithm.
 */
public Vector2 getMove() { return move; }
}

```

PieceSquareTables

```

package com.dylanwalsh.chessai.ai;

/**
 * This is a static class containing the piece square tables for each piece. These

```

```

tables are used in GameAI.
*/
public class PieceSquareTables {
    public static short[][] PAWNTABLE =
    {
        {70, 70, 70, 70, 70, 70, 70, 70},
        {50, 50, 50, 50, 50, 50, 50, 50},
        {10, 10, 20, 30, 30, 20, 10, 10},
        {5, 5, 10, 25, 25, 10, 5, 5},
        {0, 0, 0, 20, 20, 0, 0, 0},
        {5, -5, -10, 0, 0, -10, -5, 5},
        {5, 10, 10, -20, -20, 10, 10, 5},
        {0, 0, 0, 0, 0, 0, 0, 0}
    };

    public static final short[][] KNIGHTTABLE =
    {
        {-50, -40, -30, -30, -30, -30, -40, -50},
        {-40, -20, 0, 0, 0, 0, -20, -40},
        {-30, 0, 10, 15, 15, 10, 0, -30},
        {-30, 5, 15, 20, 20, 15, 5, -30},
        {-30, 0, 15, 20, 20, 15, 0, -30},
        {-30, 5, 10, 15, 15, 10, 5, -30},
        {-40, -20, 0, 5, 5, 0, -20, -40},
        {-50, -40, -30, -30, -30, -30, -40, -50}
    };

    public static final short[][] BISHOPTABLE =
    {
        {-20, -10, -10, -10, -10, -10, -10, -20},
        {-10, 0, 0, 0, 0, 0, 0, -10},
        {-10, 0, 5, 10, 10, 5, 0, -10},
        {-10, 5, 5, 10, 10, 5, 5, -10},
        {-10, 0, 10, 10, 10, 10, 0, -10},
        {-10, 10, 10, 10, 10, 10, 10, -10},
        {-10, 5, 0, 0, 0, 0, 5, -10},
        {-20, -10, -10, -10, -10, -10, -10, -20}
    };

    public static final short[][] ROOKTABLE =
    {
        {0, 0, 0, 0, 0, 0, 0, 0},
        {5, 10, 10, 10, 10, 10, 10, 5},
        {-5, 0, 0, 0, 0, 0, 0, -5},
        {-5, 0, 0, 0, 0, 0, 0, -5},
        {-5, 0, 0, 0, 0, 0, 0, -5},
        {-5, 0, 0, 0, 0, 0, 0, -5},
        {-5, 0, 0, 0, 0, 0, 0, -5},
        {0, 0, 0, 5, 5, 0, 0, 0}
    };

    public static final short[][] QUEENTABLE =
    {
        {-20, -10, -10, -5, -5, -10, -10, -20},
        {-10, 0, 0, 0, 0, 0, 0, -10},
        {-10, 0, 5, 5, 5, 5, 0, -10},
        {-5, 0, 5, 5, 5, 5, 0, -5},
        {0, 0, 5, 5, 5, 5, 0, -5},
        {-10, 5, 5, 5, 5, 5, 0, -10},
        {-10, 0, 5, 0, 0, 0, 0, -10},
        {-20, -10, -10, -5, -5, -10, -10, -20}
    };

    public static final short[][] KINGTABLE =
    {
        {-30, -40, -40, -50, -50, -40, -40, -30},
        {-30, -40, -40, -50, -50, -40, -40, -30},
        {-30, -40, -40, -50, -50, -40, -40, -30},
        {-30, -40, -40, -50, -50, -40, -40, -30},
        {-20, -30, -30, -40, -40, -30, -30, -20}
    };
}

```

```

        {-10,-20,-20,-20,-20,-20,-20,-10},
        {20, 20,  0,  0,  0,  0, 20, 20},
        {20, 30, 10,  0,  0, 10, 30, 20}
    };
    public static short[][] reverseTable(short[][] table) {
        short[][] reversedTable = new short[8][8];
        for(int i = table.length-1; i >= 0; i--) {
            reversedTable[i] = table[7-i];
        }
        return reversedTable;
    }
}

```

RelativePieceValues

```

package com.dylanwalsh.chessai.ai;

/**
 * A static class to contain the relative piece values for each piece. These values
 * are used in GameAI.
 */
public class RelativePieceValues{
    //Piece value constants
    public static final int PAWN = 100;
    public static final int KNIGHT = 320;
    public static final int BISHOP = 330;
    public static final int ROOK = 500;
    public static final int QUEEN = 900;
    public static final int KING = 20000;
}

```

com.dylanwalsh.chessai.database

DBConnection

```

package com.dylanwalsh.chessai.database;

import java.sql.*;
import java.util.ArrayList;

//TODO: Make high score stored procedure

/**
 * This is a static class that deals with all functionality concerned with
 * communicating with the database. This includes sending data
 * to be recorded in the database as well as retrieving data from the database. A
 * GameData object is returned from each method where
 * data for a game can be easily accessed.
 */
public class DBConnection {
    //Values for communicating with the database.
    private static String connectionURL;
    private static String user;
    private static String pass;

    /**
     * The values needed to connect to the database are initialized in this method.
     * This is only called once.
     */
    public static void initialize() {
        connectionURL = "jdbc:mysql://localhost:3306/chess_database";
        user = "user01";
        pass = "password1";
    }

    /**

```

```

    * A method to store details of a game in the database. The games table in the
    database is either updated or a new record is
    * added depending on whether this is a new game or an old game being re-saved.
    A new user is added to the users table depending
    * on whether this is a new user saving the game or not.
    * @param name the name of the user saving the game.
    * @param time the time elapsed for the player for the game being saved.
    * @param moves the number of moves made for the game being saved.
    * @param gameMoves a string representing the move history for the game being
    saved.
    * @param win the state of the game being saved. Win - 1, Loss - 0, Unfinished
    - 2
    * @param gameId the gameId of the game to update if this is a re-save,
    otherwise -1.
    * @return a new GameData object representing the saved game.
    */
    public static GameData saveGame(String name, int time, int moves, String
    gameMoves, int win, int gameId) {

        Connection dbConn = null;
        PreparedStatement dbStmt1 = null;
        PreparedStatement dbStmt2 = null;

        try {
            dbConn = DriverManager.getConnection(connectionURL, user, pass);

            /*
            I use 4 SQL statements in the saveGame() method, all of which are
            parametrized queries. I do this so that values
            passed into this method can be easily inserted into the queries. The
            first 2 queries are used for the users table and
            the final 2 are for the games table, though I do use a sub-query in
            query2 to access the user_id from the users table.

            The CURDATE() function in mysql returns the current date in the format
            YYYY-MM-DD.
            */
            String query1Check = "SELECT * FROM users WHERE name=?";
            String query1 =
                "INSERT INTO users(name) VALUES(?);";
            String query2 =
                "INSERT INTO games(time_stamp, time_survived, score, move_history,
            user_id, win_loss) VALUES((SELECT CURDATE()), ?, ?, ?, (SELECT id FROM users WHERE
            name = ?), ?)";
            String query2Update =
                "UPDATE games SET time_stamp=(SELECT CURDATE()), time_survived=?,
            score=?, move_history=?, win_loss=? WHERE id=?";

            dbStmt1 = dbConn.prepareStatement(query1Check);
            dbStmt1.setString(1, name);
            ResultSet dbRes1 = dbStmt1.executeQuery();
            if(!dbRes1.next() && gameId==-1) { //Name doesn't exist in users and
            not updating an older game.
                dbStmt1 = dbConn.prepareStatement(query1);
                dbStmt1.setString(1, name);
                dbStmt1.executeUpdate();
            }

            if(gameId == -1) { //Newly save game.
                dbStmt2 = dbConn.prepareStatement(query2);
                dbStmt2.setInt(1, time);
                dbStmt2.setInt(2, moves);
                dbStmt2.setString(3, gameMoves);
                dbStmt2.setString(4, name); //Sub-query to look up user id using
            name
                dbStmt2.setInt(5, win);
            } else { //Save over old game.

```

```

        dbStmt2 = dbConn.prepareStatement(query2Update);
        dbStmt2.setInt(1, time);
        dbStmt2.setInt(2, moves);
        dbStmt2.setString(3, gameMoves);
        dbStmt2.setInt(4, win);
        dbStmt2.setInt(5, gameId);
    }

    dbStmt2.executeUpdate();

} catch(SQLException e) {
    System.out.println("SQLException thrown, server may not be running!");
} finally {
    try {
        if (dbConn != null)
            dbConn.close();
        if (dbStmt1 != null)
            dbStmt1.close();
        if (dbStmt2 != null)
            dbStmt2.close();
    } catch(SQLException e2) {}
}

return getLatestGame(gameId);
}
/**
 * Retrieves latest game added/modified row in games table. If userId is -1,
 * get the row with the highest id, else get the row
 * with that id.
 * @param gameId the id of the game if it is known. Otherwise -1.
 * @return a new GameData object representing the game retrieved from the
 * database.
 */
public static GameData getLatestGame(int gameId) {
    Date timeStamp = null;
    int timeSurvived = -1;
    int score = -1;
    String moveHistory = "";
    String name = "";
    int win = -1;

    Connection dbConn = null;
    Statement dbStmt1 = null;
    PreparedStatement dbStmt2 = null;
    try {
        dbConn = DriverManager.getConnection(connectionURL, user, pass);

        String query1 = "SELECT id, time_stamp, time_survived, score,
move_history, (SELECT name FROM users WHERE id=user_id) AS name, win_loss FROM
games ORDER BY id DESC LIMIT 0, 1;";
        String query2 = "SELECT time_stamp, time_survived, score, move_history,
(SELECT name FROM users WHERE id=user_id) AS name, win_loss FROM games WHERE id=?";

        if(gameId== -1) { //Retrieve most recently added row.

            dbStmt1 = dbConn.createStatement();
            ResultSet dbRes = dbStmt1.executeQuery(query1);

            if(dbRes.next()) {
                gameId = dbRes.getInt("id");
                timeStamp = dbRes.getDate("time_stamp");
                timeSurvived = dbRes.getInt("time_survived");
                score = dbRes.getInt("score");
                moveHistory = dbRes.getString("move_history");
                name = dbRes.getString("name");
                win = dbRes.getInt("win_loss");
            }

```

```

    } else { //Retrieve row with gameId.
        dbStmt2 = dbConn.prepareStatement(query2);
        dbStmt2.setInt(1, gameId);
        ResultSet dbRes = dbStmt2.executeQuery();

        if(dbRes.next()) {
            timeStamp = dbRes.getDate("time_stamp");
            timeSurvived = dbRes.getInt("time_survived");
            score = dbRes.getInt("score");
            moveHistory = dbRes.getString("move_history");
            name = dbRes.getString("name");
            win = dbRes.getInt("win_loss");
        }
    }

} catch(SQLException e) {
    System.out.println("SQLException thrown, server may not be running!");
} finally {
    try {
        if(dbConn != null)
            dbConn.close();
        if(dbStmt1 != null)
            dbStmt1.close();
        if(dbStmt2 != null)
            dbStmt2.close();
    } catch(SQLException e2) {}
}

return new GameData(gameId, timeStamp, timeSurvived, score, moveHistory,
name, win);

}

/**
 * This method retrieves all previously saved games in the database and returns
them in the form of a list.
 * @return an ArrayList containing GameData objects.
 */
public static ArrayList<GameData> getPastGames() {

    ArrayList<GameData> games = new ArrayList<GameData>();

    Connection dbConn = null;
    Statement dbStmt = null;
    try {
        dbConn = DriverManager.getConnection(connectionURL, user, pass);

        String query = "SELECT id, time_stamp, time_survived, score,
move_history, (SELECT name FROM users WHERE id=user_id) AS name, win_loss FROM
games;";

        dbStmt = dbConn.createStatement();
        ResultSet dbRes = dbStmt.executeQuery(query);
        while(dbRes.next()) {

            int gameId = dbRes.getInt("id");
            Date timeStamp = dbRes.getDate("time_stamp");
            int timeSurvived = dbRes.getInt("time_survived");
            int score = dbRes.getInt("score");
            String moveHistory = dbRes.getString("move_history");
            String name = dbRes.getString("name");
            int win = dbRes.getInt("win_loss");

            games.add(new GameData(gameId, timeStamp, timeSurvived, score,
moveHistory, name, win));
        }
    }
}

```

```

    } catch(SQLException e) {
        System.out.println("SQLException thrown, server may not be running!");
    } finally {
        try {
            if(dbConn != null)
                dbConn.close();
            if(dbStmt != null)
                dbStmt.close();
        } catch(SQLException e2) {}
    }

    return games;
}
}
}

```

GameData

```

package com.dylanwalsh.chessai.database;

import java.sql.Date;

/**
 * Stores data about a single game.
 */
public class GameData {

    /**
     * All private fields for this class, each representing a column in the games
     table, except name, which is retrieved
     * using the user_id.
     */

    private int gameId;
    private Date timeStamp;
    private int timeSurvived;
    private int score;
    private String moveHistory;
    private String name;
    private int win;

    public GameData(int gameId, Date timeStamp, int timeSurvived, int score, String
moveHistory, String name, int win) {
        this.gameId = gameId;
        this.timeStamp = timeStamp;
        this.timeSurvived = timeSurvived;
        this.score = score;
        this.moveHistory = moveHistory;
        this.name = name;
        this.win = win;
    }

    /**
     * The following are all getter methods for the private fields of this class.
     This way I can be shore the values for the private
     * fields are not modified outside of this class.
     */

    public int getGameId() {
        return gameId;
    }

    public Date getTimeStamp() {
        return timeStamp;
    }

    public int getTimeSurvived() {
        return timeSurvived;
    }
}

```

```

    }
    public int getScore() {
        return score;
    }
    public String getMoveHistory() {
        return moveHistory;
    }
    public String getName() {
        return name;
    }
    public int getWin() {
        return win;
    }
}

```

com.dylanwalsh.chessai.entities

com.dylanwalsh.chessai.entities.chesspieces

Bishop

```

package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class Bishop extends com.dylanwalsh.chessai.entities.ChessPiece {

    public Bishop(Pieces type, int startX, int startY){ super(type, startX,
startY); }
    @Override
    public void generatePositions(com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        //Up-right diagonal.
        for(int y=(int)getPosition().y+1, x=(int)getPosition().x+1; y<board.length
&& x<board[(int)getPosition().y].length; y++, x++) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //Otherwise next iteration.
        }
        //Up-left diagonal.
        for(int y=(int)getPosition().y+1, x=(int)getPosition().x-1; y<board.length
&& x>=0; y++, x--) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //Otherwise next iteration.
        }
        //Down-right diagonal.
        for(int y=(int)getPosition().y-1, x=(int)getPosition().x+1; y>=0 &&
x<board[(int)getPosition().y].length; y--, x++) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //Otherwise next iteration.
        }
        //Down-left diagonal.
        for(int y=(int)getPosition().y-1, x=(int)getPosition().x-1; y>=0 && x>=0;
y--, x--) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //Otherwise next iteration.
        }

        //Validate positions.
        validate(board);
    }
}

```


King

```
package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class King extends com.dylanwalsh.chessai.entities.ChessPiece {
    //References to castle conditions in Board so that king can know when to add
    castle position to its availablePositions.
    private boolean canCastleLeft;
    private boolean canCastleRight;
    private Rook castleLeftRook;
    private Rook castleRightRook;
    private Vector2 castleLeftPosition;
    private Vector2 castleRightPosition;
    //Variable to track whether this King has moved.
    private boolean moved = false;

    public King(Pieces type, int startX, int startY, Rook castleLeftRook, Rook
    castleRightRook) {
        super(type, startX, startY);
        this.castleLeftRook = castleLeftRook;
        this.castleRightRook = castleRightRook;
        canCastleLeft = false;
        canCastleRight = false;
    }
    /**
     * A method that returns true if this king is in check. The method calls
     isCheckingKing() of every ChessPiece
     * on the board until one returns true.
     * @param board The 2D board array.
     * @return true or false, depending on whether this king is currently in check
     */
    public boolean isInCheck(com.dylanwalsh.chessai.entities.ChessPiece[][] board)
    {
        for(com.dylanwalsh.chessai.entities.ChessPiece[] row : board) {
            for(com.dylanwalsh.chessai.entities.ChessPiece piece : row) {
                if(piece != null && piece.getPiece().toString().charAt(0) !=
                getPiece().toString().charAt(0) && piece.isCheckingKing()) {
                    return true;
                }
            }
        }
        return false;
    }
    /**
     * This method is only called once during the initialisation of the board. It
     sets the two positions this King will
     * be able to move to if a castling move were to be performed.
     * @param lx the x position if this king were to castle left.
     * @param ly the y position if this king were to castle left.
     * @param rx the x position if this king were to castle right.
     * @param ry the y position if this king were to castle right.
     */
    public void setCastlingPositions(int lx, int ly, int rx, int ry) {
        castleLeftPosition = new Vector2(lx, ly);
    }
}
```

```

        castleRightPosition = new Vector2(rx, ry);
    }
    public boolean canCastleLeft() {
        return canCastleLeft;
    }
    public boolean canCastleRight() {
        return canCastleRight;
    }
    public Rook getCastleLeftRook() {
        return castleLeftRook;
    }
    public Rook getCastleRightRook() {
        return castleRightRook;
    }
    public Vector2 getCastleLeftPosition() {
        return castleLeftPosition;
    }
    public Vector2 getCastleRightPosition() {
        return castleRightPosition;
    }
    public void setCanCastleLeft(boolean canCastleLeft) {
        this.canCastleLeft = canCastleLeft;
    }
    public void setCanCastleRight(boolean canCastleRight) {
        this.canCastleRight = canCastleRight;
    }
    public void setMoved() {
        moved = true;
    }
    public boolean hasMoved() {
        return moved;
    }
    @Override
    public void generatePositions(com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        //Left column.
        for(int i=-1; i<2; i++) { tempPos.add(new Vector2(getPosition().x-1,
getPosition().y+i)); }
        //Right column.
        for(int i=-1; i<2; i++) { tempPos.add(new Vector2(getPosition().x+1,
getPosition().y+i)); }
        //Top and bottom.
        tempPos.add(new Vector2(getPosition().x, getPosition().y+1));
        tempPos.add(new Vector2(getPosition().x, getPosition().y-1));

        //Add the castling positions if possible.
        if(canCastleLeft) {
            availablePositions.add(new Vector2((int)castleLeftPosition.x,
(int)castleLeftPosition.y));
        }
        if(canCastleRight) {
            availablePositions.add(new Vector2((int)castleRightPosition.x,
(int)castleRightPosition.y));
        }

        //Validate positions.
        validate(board);
    }
}

```

Knight

```
package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class Knight extends com.dylanwalsh.chessai.entities.ChessPiece {

    public Knight(Pieces type, int startX, int startY){
        super(type, startX, startY);
    }
    @Override
    public void generatePositions (com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        //Right of knight.
        tempPos.add(new Vector2(getPosition().x+1, getPosition().y+2));
        tempPos.add(new Vector2(getPosition().x+1, getPosition().y-2));

        tempPos.add(new Vector2(getPosition().x+2, getPosition().y+1));
        tempPos.add(new Vector2(getPosition().x+2, getPosition().y-1));

        //Left of knight.
        tempPos.add(new Vector2(getPosition().x-1, getPosition().y+2));
        tempPos.add(new Vector2(getPosition().x-1, getPosition().y-2));

        tempPos.add(new Vector2(getPosition().x-2, getPosition().y+1));
        tempPos.add(new Vector2(getPosition().x-2, getPosition().y-1));

        //Validate positions.
        validate(board);
    }
}
```

Pawn

```
package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class Pawn extends com.dylanwalsh.chessai.entities.ChessPiece {

    public Pawn(Pieces type, int startX, int startY){
        super(type, startX, startY);
    }
    @Override
    public void generatePositions (com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        switch(getPiece()) {
            case BPAWN:
                tempPos.add(new Vector2(getPosition().x, getPosition().y-1));
                //Check position - determine if made first move.
                if(getPosition().y == 6) { //Index 6 - hasn't moved.
                    if(board[(int)getPosition().y-1][(int)getPosition().x] == null)
                        tempPos.add(new Vector2(getPosition().x, getPosition().y-
2)); //two down
                }
            }
        }
    }
}
```

```

        }
        break;
    case WPAWN:
        tempPos.add(new Vector2(getPosition().x, getPosition().y+1));
        //Check position - determine if made first move.
        if(getPosition().y == 1) { //Index 1 - hasn't moved
            if(board[(int)getPosition().y+1][(int)getPosition().x] == null)
                tempPos.add(new Vector2(getPosition().x,
getPosition().y+2)); //two up
            }
            break;
        }
    }

    //Validate positions.
    validate(board);
}

public char promote(com.dylanwalsh.chessai.entities.ChessPiece[][] board, char
promoteTo) {
    //For black, 50% chance of Queen or Knight.
    if(getPiece().toString().charAt(0) == 'B' && promoteTo=='-') {
        char[] p = new char[]{'Q', 'K'};
        promoteTo = p[(int)Math.round(Math.random())];
    }

    switch(promoteTo) {
        case 'Q':
            board[(int)getPosition().y][(int)getPosition().x] = new
Queen(getPiece()==Pieces.WPAWN?Pieces.WQUEEN:Pieces.BQUEEN, (int)getPosition().x,
(int)getPosition().y);
            dispose(); //Dispose this pawn.
            break;
        case 'K':
            board[(int)getPosition().y][(int)getPosition().x] = new
Knight(getPiece()==Pieces.WPAWN?Pieces.WKNIGHT:Pieces.BKNIGHT,
(int)getPosition().x, (int)getPosition().y);
            dispose(); //Dispose this pawn.
            break;
        case 'B':
            board[(int)getPosition().y][(int)getPosition().x] = new
Bishop(getPiece()==Pieces.WPAWN?Pieces.WBISHOP:Pieces.BBISHOP,
(int)getPosition().x, (int)getPosition().y);
            dispose(); //Dispose this pawn.
            break;
        case 'R':
            board[(int)getPosition().y][(int)getPosition().x] = new
Rook(getPiece()==Pieces.WPAWN?Pieces.WROOK:Pieces.BROOK, (int)getPosition().x,
(int)getPosition().y, 'N');
            dispose(); //Dispose this pawn.
            break;
    }
    //Call generatePositions on newly added piece.
    board[(int)getPosition().y][(int)getPosition().x].generatePositions(board);

    return promoteTo;
}
}

```

Queen

```

package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class Queen extends com.dylanwalsh.chessai.entities.ChessPiece {
    public Queen(Pieces type, int startX, int startY){
        super(type, startX, startY);
    }
}

```

```

    }

    @Override
    public void generatePositions(com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        //Up.
        for(int y = (int)getPosition().y+1; y<board.length; y++) {
            tempPos.add(new Vector2(getPosition().x, y));
            if(board[y][(int)getPosition().x] != null) break; //otherwise next
iteration
        }
        //Down.
        for(int y = (int)getPosition().y-1; y>=0; y--) {
            tempPos.add(new Vector2(getPosition().x, y));
            if(board[y][(int)getPosition().x] != null) break; //otherwise next
iteration
        }
        //Right.
        for(int x = (int)getPosition().x+1; x<board[(int)getPosition().y].length;
x++) {
            tempPos.add(new Vector2(x, getPosition().y));
            if(board[(int)getPosition().y][x] != null) break; //otherwise next
iteration
        }
        //Left.
        for(int x = (int)getPosition().x-1; x>=0; x--) {
            tempPos.add(new Vector2(x, getPosition().y));
            if(board[(int)getPosition().y][x] != null) break; //otherwise next
iteration
        }
        //Up-right diagonal.
        for(int y=(int)getPosition().y+1, x=(int)getPosition().x+1; y<board.length
&& x<board[(int)getPosition().y].length; y++, x++) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //otherwise next iteration
        }
        //Up-left diagonal.
        for(int y=(int)getPosition().y+1, x=(int)getPosition().x-1; y<board.length
&& x>=0; y++, x--) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //otherwise next iteration
        }
        //Down-right.
        for(int y=(int)getPosition().y-1, x=(int)getPosition().x+1; y>=0 &&
x<board[(int)getPosition().y].length; y--, x++) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //otherwise next iteration
        }
        //Down-left.
        for(int y=(int)getPosition().y-1, x=(int)getPosition().x-1; y>=0 && x>=0;
y--, x--) {
            tempPos.add(new Vector2(x, y));
            if(board[y][x] != null) break; //otherwise next iteration
        }

        //Validate positions.
        validate(board);
    }
}

```

Rook

```
package com.dylanwalsh.chessai.entities.chesspieces;

import com.badlogic.gdx.math.Vector2;

public class Rook extends com.dylanwalsh.chessai.entities.ChessPiece {
    private Vector2 castlePosition;
    //Left or right rook?
    private char rookType; //Left or right -> 'L', 'R', for pawns promoting to new
    rook -> 'N' (this could be any value).
    //Variable to track whether this Rook has moved.
    private boolean moved = false;

    public Rook(Pieces type, int startX, int startY, char rookType){
        super(type, startX, startY);
        this.rookType = rookType;
    }
    /**
     * This method is only called once during the initialisation of the board. It
     sets the position this Rook will be able to move
     * to if a castling move were to be performed.
     * @param x the x position if this rook were to castle.
     * @param y the y position if this king were to castle.
     */
    public void setCastlingPosition(int x, int y) {
        castlePosition = new Vector2(x, y);
    }
    public char getType() { return rookType; }
    public void setMoved() {
        moved = true;
    }
    public boolean hasMoved() {
        return moved;
    }
    public Vector2 getCastlePosition() {
        return castlePosition;
    }
    @Override
    public void generatePositions(com.dylanwalsh.chessai.entities.ChessPiece[][]
board) {
        //Clear availablePositions and tempPos.
        availablePositions.clear();
        tempPos.clear();

        //Add positions to tempPos (excluding current position).
        //Up.
        for(int y = (int)getPosition().y+1; y<board.length; y++) {
            tempPos.add(new Vector2(getPosition().x, y));
            if(board[y][(int)getPosition().x] != null) break; //otherwise next
iteration
        }
        //Down.
        for(int y = (int)getPosition().y-1; y>=0; y--) {
            tempPos.add(new Vector2(getPosition().x, y));
            if(board[y][(int)getPosition().x] != null) break; //otherwise next
iteration
        }
        //Right.
        for(int x = (int)getPosition().x+1; x<board[(int)getPosition().y].length;
x++) {
            tempPos.add(new Vector2(x, getPosition().y));
            if(board[(int)getPosition().y][x] != null) break; //otherwise next
iteration
        }
        //Left.
        for(int x = (int)getPosition().x-1; x>=0; x--) {
            tempPos.add(new Vector2(x, getPosition().y));

```

```

        if(board[(int)getPosition().y][x] != null) break; //otherwise next
iteration
    }

    //Validate positions.
    validate(board);
}
}

```

ChessPiece

```

package com.dylanwalsh.chessai.entities;

import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.utils.Disposable;
import com.dylanwalsh.chessai.entities.chesspieces.King;
import com.dylanwalsh.chessai.entities.chesspieces.Pawn;
import com.dylanwalsh.chessai.input.GameInput;
import com.dylanwalsh.chessai.screens.HUD;
import com.dylanwalsh.chessai.util.PieceMovements;

import java.util.ArrayList;

/**
 * This is an abstract class containing all functionality for a chess piece. Piece
specific functionality is implemented in child
 * classes inheriting from ChessPiece. This class implements Disposable so that it
can dispose chess piece textures.
 */
public abstract class ChessPiece implements Disposable{

    /**
 * This is a enum storing all different types of pieces. Every child class of
ChessPiece must call the ChessPiece
 * constructor to initialize the piece value, using one of these enum values.
 */
    public enum Pieces { //Implicitly static.
        WBISHOP, WKING, WKNIGHT, WPAWN, WQUEEN, WROOK,
        BBISHOP, BKING, BKNIGHT, BPAWN, BQUEEN, BROOK
    }

    private final Pieces piece;
    private Vector2 position;
    //Temporary positions added to in generatePositions, validate() method checks
if these positions are valid
    //and then adds to availablePositions.
    protected ArrayList<Vector2> tempPos;
    //Positions that this piece can move to. Will not include current position.
    protected ArrayList<Vector2> availablePositions;
    //Contains list of potential takes for this piece.
    private ArrayList<Vector2> potentialTakes;
    private final Texture pieceTexture;
    private boolean checkingKing;

    /**
 * The ChessPiece constructor, a call to super from the child constructor
initializes the chess piece's type, and start position
 * and assigns the piece the appropriate texture.
 * @param piece the type of piece.
 * @param startX the starting x value of this piece.
 * @param startY the starting y value of this piece.
 */
    public ChessPiece(Pieces piece, int startX, int startY) {
        this.piece = piece;
        position = new Vector2().set(startX, startY);
        tempPos = new ArrayList<Vector2>();
    }
}

```

```

availablePositions = new ArrayList<Vector2>();
potentialTakes = new ArrayList<Vector2>();
checkingKing = false;

switch(piece) {
    case BBISHOP:
        pieceTexture = new Texture("chess_pieces/bbishop.png");
        break;
    case WBISHOP:
        pieceTexture = new Texture("chess_pieces/wbishop.png");
        break;
    case BKING:
        pieceTexture = new Texture("chess_pieces/bking.png");
        break;
    case WKING:
        pieceTexture = new Texture("chess_pieces/wking.png");
        break;
    case BKNIGHT:
        pieceTexture = new Texture("chess_pieces/bknight.png");
        break;
    case WKNIGHT:
        pieceTexture = new Texture("chess_pieces/wknight.png");
        break;
    case BQUEEN:
        pieceTexture = new Texture("chess_pieces/bqueen.png");
        break;
    case WQUEEN:
        pieceTexture = new Texture("chess_pieces/wqueen.png");
        break;
    case BROOK:
        pieceTexture = new Texture("chess_pieces/brook.png");
        break;
    case WROOK:
        pieceTexture = new Texture("chess_pieces/wrook.png");
        break;
    case BPAWN:
        pieceTexture = new Texture("chess_pieces/bpawn.png");
        break;
    case WPAWN:
        pieceTexture = new Texture("chess_pieces/wpawn.png");
        break;
    default:
        pieceTexture = new Texture("chess_pieces/wpawn.png");
        break;
}

/**
 * @return the piece texture.
 */
public Texture getPieceTexture() {
    return pieceTexture;
}

/**
 *
 * @param posX
 * @param posY
 * @param board
 * @param hud
 * @param in
 * @param promotePawn The piece this pawn will be promoted to if it is known
(this is a loaded game). Otherwise '-'.
 * @return
 */
public boolean moveTo(int posX, int posY, ChessPiece[][] board, HUD hud,
GameInput in, char promotePawn) {
    //Look over all the available positions for this chess piece. Here I look
over each item in the list and compare it to
    //the posX, posY position passed into this method.
    for (Vector2 pos : availablePositions) {

```



```

        if (pos.x == posX && pos.y == posY) {
            //Valid move.
            PieceMovements.move(board, this, posX, posY);
            if (getFriendlyKing(board).isInCheck(board)) {
                PieceMovements.undo();
                return false;
            }
            //Record last game move.
            PieceMovements.recordLastMove();
            //Check if pawn can be promoted.
            switch (getPiece()) {
                case BPAWN:
                    if (posY == 0) {
                        PieceMovements.promoteBlackPawn(((Pawn) this),
promotePawn);
                    }
                    break;
                case WPAWN:
                    if (posY == 7) {
                        PieceMovements.promoteWhitePawn(((Pawn) this),
promotePawn, in);
                    }
                    break;
            }

            return true;
        }
    }
    for (Vector2 pos : potentialTakes) {
        if (pos.x == posX && pos.y == posY) {
            //Valid move.
            ChessPiece referenceToPiece = PieceMovements.move(board, this,
posX, posY);

            if (getFriendlyKing(board).isInCheck(board)) {
                PieceMovements.undo();
                return false;
            }

            //Add taken piece to hud.
            hud.addTakenPiece(referenceToPiece);
            referenceToPiece.dispose(); //Dispose the taken piece.

            //Record last game move.
            PieceMovements.recordLastMove();
            //Check if pawn can be promoted.
            switch (getPiece()) {
                case BPAWN:
                    if (posY == 0)
                        PieceMovements.promoteBlackPawn(((Pawn) this),
promotePawn);
                    break;
                case WPAWN:
                    if (posY == 7) {
                        PieceMovements.promoteWhitePawn(((Pawn) this),
promotePawn, in);
                    }
                    break;
            }
            return true;
        }
    }
    return false;
}

public King getFriendlyKing(ChessPiece[][] board) {
    for(ChessPiece[] row : board) {
        for(ChessPiece piece : row) {
            if(piece != null)
                if(piece.getPiece() == Pieces.BKING || piece.getPiece() ==
Pieces.WKING) //It is a King.

```

```

        if(piece.getPiece().toString().charAt(0) ==
getPiece().toString().charAt(0)) {//Of the same colour.
            return (King) piece;
        }
    }
}

return null;
}
/**
 * Returns the state of the checkingKing flag.
 *
 * @return true of false, depending on whether this piece is checking the
opponent's king.
 */
public boolean isCheckingKing() {
    return checkingKing;
}
/**
 * A method called automatically by validate() after every move which sets the
checkingKing flag to true or false depending
 * on whether the potentialTakes contains the opponent's king.
 */
private void setCheckingKing(ChessPiece[][] board) {
    //PotentialTakes will only contain opponent pieces, and therefore won't
contain this sides king.
    for(Vector2 piece : potentialTakes) {
        if (board[(int) piece.y][(int) piece.x].getPiece() == Pieces.BKING ||
board[(int) piece.y][(int) piece.x].getPiece() == Pieces.WKING) {
            checkingKing = true;
            return;
        }
    }
    checkingKing = false;
}
/**
 * This method validates that any position added by a piece is in the range of
the board and that it doesn't already contain a piece.
 * For pieces other than the pawns, this method will validate that a piece can
be taken if it is within its move set and is of the
 * opposite colour. For the pawn pieces, this method validates that the
position a pawn can move to as part of its attacking
 * moves contains a piece and is of the opposite colour.
 *
 * An implementation of this abstract class will call it's generatePositions(),
which calls this method which in turn calls
 * setCheckingKing().
 * generatePositions() -> validate() -> setCheckingKing()
 */
protected void validate(ChessPiece[][] board) {
    //Clear the potentialTakes array.
    potentialTakes.clear();
    //Pawns.
    if(getPiece() == Pieces.BPAWN) { //Black pawn.
        int y1 = (int)getPosition().y-1;
        int x1 = (int)getPosition().x+1;
        if(x1 < 8 && x1 >= 0 && y1 < 8 && y1 >= 0)
            if(board[y1][x1] != null)
                if(board[y1][x1].getPiece().toString().charAt(0) !=
getPiece().toString().charAt(0)) //Opposite colour.
                    potentialTakes.add(new Vector2(x1, y1));

        int y2 = (int)getPosition().y-1;
        int x2 = (int)getPosition().x-1;
        if(x2 < 8 && x2 >= 0 && y2 < 8 && y2 >= 0)
            if(board[y2][x2] != null)
                if(board[y2][x2].getPiece().toString().charAt(0) !=
getPiece().toString().charAt(0)) //Opposite colour.

```

```

        potentialTakes.add(new Vector2(x2, y2));
    }

    if(getPiece() == Pieces.WPAWN) { //White pawn.
        int y1 = (int)getPosition().y+1;
        int x1 = (int)getPosition().x+1;
        if(x1 < 8 && x1 >= 0 && y1 < 8 && y1 >= 0)
            if(board[y1][x1] != null)
                if(board[y1][x1].getPiece().toString().charAt(0) !=
getPiece().toString().charAt(0)) //Opposite colour.
                    potentialTakes.add(new Vector2(x1, y1));

        int y2 = (int)getPosition().y+1;
        int x2 = (int)getPosition().x-1;
        if(x2 < 8 && x2 >= 0 && y2 < 8 && y2 >= 0)
            if(board[y2][x2] != null)
                if(board[y2][x2].getPiece().toString().charAt(0) !=
getPiece().toString().charAt(0)) //Opposite colour.
                    potentialTakes.add(new Vector2(x2, y2));
    }

    for(Vector2 pos : tempPos) {
        if(pos.x < 8 && pos.x >= 0 && pos.y < 8 && pos.y >= 0) { //They are
positions on the board.
            if(board[(int)pos.y][(int)pos.x] == null) { //There are no pieces
in this position.
                availablePositions.add(pos);
            } else { //If not null, piece can be taken if opposite colour.
                if(getPiece() != Pieces.WPAWN && getPiece() != Pieces.BPAWN) {
//Forward moves for pawn cannot take a piece.
                    if (board[(int) pos.y][(int)
pos.x].getPiece().toString().charAt(0) != getPiece().toString().charAt(0)) {
                        potentialTakes.add(pos);
                    }
                }
            }
        }
    }

    //Check if piece is now checking the king.
    setCheckingKing(board);
}
/**
 * A method that retrieves all available positions and potential takes for this
piece.
 * @return an ArrayList of all moves for this piece.
 */
public ArrayList<Vector2> getAllPotentialMoves() {
    ArrayList<Vector2> returnList = new ArrayList<Vector2>(potentialTakes);
    returnList.addAll(availablePositions);
    return returnList;
}
/**
 * The abstract method implemented by child classes to generate piece specific
movements.
 * @param board the 2D board array.
 */
public abstract void generatePositions(ChessPiece[][] board);
public Pieces getPiece() {
    return piece;
}
public Vector2 getPosition() {
    return position;
}
public void setPosition(Vector2 position) {

```

```

        this.position = position;
    }
    @Override
    public void dispose() {
        pieceTexture.dispose();
    }
}

```

com.dylanwalsh.chessai.input

GameInput

```

package com.dylanwalsh.chessai.input;

import com.badlogic.gdx.Input;
import com.badlogic.gdx.InputProcessor;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.math.Vector3;
import com.dylanwalsh.chessai.entities.ChessPiece;
import com.dylanwalsh.chessai.entities.chesspieces.Pawn;
import com.dylanwalsh.chessai.screens.HUD;

import java.util.ArrayList;

/**
 * The class that deals with all game input functionality. It implements the
 * InputProcessor interface (InputProcessor is part of
 * LibGDX).
 */
public class GameInput implements InputProcessor {
    private int tileSize;
    private int boardSize;
    private Vector2 refHoverTile;
    private ArrayList<Vector2> refTileSelectionPositions;
    private OrthographicCamera refCam;
    private char promotePawnTo; //Q, R, B, K.
    private ChessPiece[][] refBoard;
    private boolean gameOver;
    private HUD refHud;

    public GameInput(int tileSize, int boardSize, Vector2 refHoverTile,
        ArrayList<Vector2> refTileSelectionPositions, OrthographicCamera refCam,
        ChessPiece[][] refBoard, HUD refHud) {
        this.tileSize = tileSize;
        this.boardSize = boardSize;
        this.refHoverTile = refHoverTile;
        this.refTileSelectionPositions = refTileSelectionPositions;
        this.refCam = refCam;
        this.refBoard = refBoard;
        this.refHud = refHud;
        promotePawnTo = 'Q'; //Assume promotion to queen.
        gameOver = false;
    }
    @Override
    public boolean keyDown(int keycode) {
        return false;
    }
    @Override
    public boolean keyUp(int keycode) {
        if(keycode == Input.Keys.Q) {
            if(promotePawnTo != 'Q') {
                promotePawnTo = 'Q';
                refHud.addGameMessage("Next pawn promotion: Queen");
            }
        }
        else if(keycode == Input.Keys.R) {

```

```

        if(promotePawnTo != 'R') {
            promotePawnTo = 'R';
            refHud.addGameMessage("Next pawn promotion: Rook");
        }
    }
    else if(keycode == Input.Keys.B) {
        if(promotePawnTo != 'B') {
            promotePawnTo = 'B';
            refHud.addGameMessage("Next pawn promotion: Bishop");
        }
    }
    else if(keycode == Input.Keys.K) {
        if(promotePawnTo != 'K') {
            promotePawnTo = 'K';
            refHud.addGameMessage("Next pawn promotion: Knight");
        }
    }
}

return true;
}

public char promotePawn(Pawn pawn, char promotePawn) {
    if(promotePawn!='-')
        promotePawnTo = promotePawn;

    return pawn.promote(refBoard, promotePawnTo);
}

@Override
public boolean keyTyped(char character) {
    return false;
}

@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
    return false;
}

@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
    if(!gameOver) {
        Vector3 worldMouseCoords = refCam.unproject(new Vector3(screenX,
screenY, 0));
        if(worldMouseCoords.x >= 0 && worldMouseCoords.x <= tileSize*boardSize
&& worldMouseCoords.y >= 0 && worldMouseCoords.y <=
tileSize*boardSize) {

            float xCoord = (float) Math.floor((worldMouseCoords.x / (tileSize *
boardSize)) * boardSize);
            float yCoord = (float) Math.floor((worldMouseCoords.y / (tileSize *
boardSize)) * boardSize);

            //Add the tile coordinates to tileSelectionPositions ArrayList in
Board.
            refTileSelectionPositions.add(new Vector2(xCoord, yCoord));

        }
        return true;
    } else {
        return false;
    }
}

}

@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
    return false;
}
}

```

```

@Override
public boolean mouseMoved(int screenX, int screenY) {

    if(!gameOver) {
        Vector3 worldMouseCoords = refCam.unproject(new Vector3(screenX,
screenY, 0));

        if(worldMouseCoords.x >= 0 && worldMouseCoords.x <= tileSize*boardSize
&& worldMouseCoords.y >= 0 && worldMouseCoords.y <=
tileSize*boardSize) {
            refHoverTile.x = (float) Math.floor( ( worldMouseCoords.x /
(tileSize*boardSize) ) * boardSize );
            refHoverTile.y = (float) Math.floor( ( worldMouseCoords.y /
(tileSize*boardSize) ) * boardSize );
        } else {
            refHoverTile.set(-1, -1);
        }

        return true;
    } else {
        return false;
    }
}
@Override
public boolean scrolled(int amount) {
    return false;
}
public void setGameOver() {
    gameOver = true;
}
}

```

com.dylanwalsh.chessai.screens

Board

```

package com.dylanwalsh.chessai.screens;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.utils.Disposable;
import com.dylanwalsh.chessai.ai.GameAI;
import com.dylanwalsh.chessai.entities.ChessPiece;
import com.dylanwalsh.chessai.entities.chesspieces.Bishop;
import com.dylanwalsh.chessai.entities.chesspieces.King;
import com.dylanwalsh.chessai.entities.chesspieces.Knight;
import com.dylanwalsh.chessai.entities.chesspieces.Pawn;
import com.dylanwalsh.chessai.entities.chesspieces.Queen;
import com.dylanwalsh.chessai.entities.chesspieces.Rook;
import com.dylanwalsh.chessai.input.GameInput;
import com.dylanwalsh.chessai.util.PieceMovements;

import java.util.ArrayList;

/**
 * The Board class encapsulates all functionality to do with the chess board. It
 * contains all ChessPiece objects. It implements
 * the Disposable interface so that the tiles and pieces can be disposed.
 */
public class Board implements Disposable{
    private static char turnFlag = 'W'; //W - white(player's turn), B - black(AI's
turn), E - end of game.
    private boolean gameOver;
    //Game AI.
}

```

```

private GameAI gameAI;
//Reference to the kings and rooks on the board.
private King whiteKing;
private King blackKing;
private Rook leftWhiteRook;
private Rook rightWhiteRook;
private Rook leftBlackRook;
private Rook rightBlackRook;
//Input.
private GameInput gameInput;
private Vector2 hoverTile;
//Holds x, y coordinates of selected tiles - update method checks which pieces
are in that tile.
private ArrayList<Vector2> tileSelectionPositions;
private SpriteBatch sb;
private Texture[] tiles;
private String[] boardColumns;
private HUD hud;
//2D board array of ChessPieces.
//First element of a 2D array is top left, but when it is draw it is * by index
of item and LibGDX origin is at bottom left.
private ChessPiece[][] board;
private final int tileSize;
private final int boardSize;

public Board(SpriteBatch sb, OrthographicCamera refCam, HUD hud) {
    this.hud = hud;
    tileSize = 20;
    boardSize = 8;
    this.sb = sb;
    board = new ChessPiece[boardSize][boardSize]; //Empty values are null.

    //Input.
    tileSelectionPositions = new ArrayList<Vector2>(2); //0 - first selected
tile, 1 - second selected tile.
    hoverTile = new Vector2().set(-1, -1);
    gameInput = new GameInput(tileSize, boardSize, hoverTile,
tileSelectionPositions, refCam, board, hud);

    //Pieces.
    //White pieces.
    for(int i=0; i<8; i++) { //Pawns.
        board[1][i] = new Pawn(ChessPiece.Pieces.WPAWN, i, 1);
    }

    board[0][3] = new Queen(ChessPiece.Pieces.WQUEEN, 3, 0);
    board[0][2] = new Bishop(ChessPiece.Pieces.WBISHOP, 2, 0);
    board[0][5] = new Bishop(ChessPiece.Pieces.WBISHOP, 5, 0);
    board[0][0] = new Rook(ChessPiece.Pieces.WROOK, 0, 0, 'L'); //Bottom left
Rook.
    leftWhiteRook = (Rook)board[0][0];
    leftWhiteRook.setCastlingPosition(3, 0);

    board[0][7] = new Rook(ChessPiece.Pieces.WROOK, 7, 0, 'R'); //Bottom right
Rook.
    rightWhiteRook = (Rook)board[0][7];
    rightWhiteRook.setCastlingPosition(5, 0);

    board[0][4] = new King(ChessPiece.Pieces.WKING, 4, 0, leftWhiteRook,
rightWhiteRook); //White king.
    whiteKing = (King)board[0][4];
    whiteKing.setCastlingPositions(2, 0, 6, 0);

    board[0][1] = new Knight(ChessPiece.Pieces.WKNIGHT, 1, 0);
    board[0][6] = new Knight(ChessPiece.Pieces.WKNIGHT, 6, 0);

    //Black pieces.
    for(int i=0; i<8; i++) { //Pawns.

```

```

        board[6][i] = new Pawn(ChessPiece.Pieces.BPAWN, i, 6);
    }

    board[7][3] = new Queen(ChessPiece.Pieces.BQUEEN, 3, 7);
    board[7][2] = new Bishop(ChessPiece.Pieces.BBISHOP, 2, 7);
    board[7][5] = new Bishop(ChessPiece.Pieces.BBISHOP, 5, 7);
    board[7][0] = new Rook(ChessPiece.Pieces.BROOK, 0, 7, 'L'); //Top left
Rook.
    leftBlackRook = (Rook)board[7][0];
    leftBlackRook.setCastlingPosition(3, 7);

    board[7][7] = new Rook(ChessPiece.Pieces.BROOK, 7, 7, 'R'); //Top right
Rook.
    rightBlackRook = (Rook)board[7][7];
    rightBlackRook.setCastlingPosition(5, 7);

    board[7][4] = new King(ChessPiece.Pieces.BKING, 4, 7, leftBlackRook,
rightBlackRook); //Black king.
    blackKing = (King)board[7][4];
    blackKing.setCastlingPositions(2, 7, 6, 7);

    board[7][1] = new Knight(ChessPiece.Pieces.BKNIGHT, 1, 7);
    board[7][6] = new Knight(ChessPiece.Pieces.BKNIGHT, 6, 7);

    //Initial generation of piece positions.
    for(ChessPiece[] row : board) {
        for(ChessPiece item : row) {
            if(item != null) {
                item.generatePositions(board);
            }
        }
    }

    tiles = new Texture[5];
    tiles[0] = new Texture(Gdx.files.internal("tiles/2.png")); //Darker.
    tiles[1] = new Texture(Gdx.files.internal("tiles/1.png")); //Lighter.
    tiles[2] = new Texture(Gdx.files.internal("tiles/hover.png")); //Hover.
    tiles[3] = new Texture(Gdx.files.internal("tiles/move.png")); //Move.
    tiles[4] = new Texture(Gdx.files.internal("tiles/3.png")); //Board
background.
    boardColumns = new String[] {"A", "B", "C", "D", "E", "F", "G", "H"};

    //Game AI.
    gameAI = new GameAI();

    //Reset moves in move stack.
    PieceMovements.resetMoves();

}

/**
 * The main update method of the game.
 * @return a boolean value to indicate whether the game has finished yet or
not.
 */
private boolean update() {
    switch(turnFlag) {
        case 'W':
            //Player's turn - check 1st and 2nd item in pieceSelections.
            try {
                ChessPiece piece1 =
board[(int)tileSelectionPositions.get(0).y][(int)tileSelectionPositions.get(0).x];

                //a piece has been selected. First mouse up.

                if(piece1 != null && piece1.getPiece().toString().charAt(0) ==

```



```

'W')
        checkAndMove(piece1, (int)tileSelectionPositions.get(1).x,
(int)tileSelectionPositions.get(1).y, '-');
        //checkAndMove will only execute on second mouse up.

        //Clear positions from tileSelectionPositions.
        if(tileSelectionPositions.size() >= 2) {
            tileSelectionPositions.clear();
        }

    } catch(IndexOutOfBoundsException e) {
        //No tile coordinates have been added to tileSelectionPositions
yet - do nothing.
    }
    break;
case 'B':
    gameAI.calculateBestMove(board);
    checkAndMove(gameAI.getPiece(), (int)gameAI.getMove().x,
(int)gameAI.getMove().y, '-');
    break;
}

//Check for checkmate.
if(isInCheckMate('W')) { //Black won
    hud.addGameMessage("Black has Won!");
    hud.setGameOver(0);
    return true;
}
if(isInCheckMate('B')) { //White won
    hud.addGameMessage("White has Won!");
    hud.setGameOver(1);
    return true;
}

return false;
}
/**
 * This method checks whether the given ChessPiece can be moved to the given
position(x, y) on the board. It calls moveTo on the
 * piece. If it was successful, moveTo returned true, the kings castling
conditions are set, the game moves so far are handed over
 * to the hud, the turn is changed and the user is prompted about the move made
by the AI and if they are in check.
 *
 * @param piece the ChessPiece to move
 * @param x the x-coordinate to move to
 * @param y the y-coordinate to move to
 * @param promotePawn the piece to promote the pawn to if this move is part of
the loaded game
 * @return will return true if move was successful and turnFlag changed
 */
private boolean checkAndMove(ChessPiece piece, int x, int y, char promotePawn)
{
    //gameInput used for by PieceMovements for pawn promotion.
    if(piece != null && piece.moveTo(x, y, board, hud, gameInput, promotePawn))
{//If it can move, it would have.
        //Update board array, at this point, piece would have updated it's
position vector.

        //If it is a Rook or King, set it's moved field to true.
        if(piece.getPiece() == ChessPiece.Pieces.WROOK || piece.getPiece() ==
ChessPiece.Pieces.BROOK)
            ((Rook)piece).setMoved();
        else if(piece.getPiece() == ChessPiece.Pieces.WKING || piece.getPiece()
== ChessPiece.Pieces.BKING)
            ((King)piece).setMoved();

```

```

        //Check castling conditions for next turn.
        whiteKing.setCanCastleLeft(checkCastlingCondition(whiteKing,
leftWhiteRook, new Vector2(1, 0), new Vector2(2, 0), new Vector2(3, 0)));
        whiteKing.setCanCastleRight(checkCastlingCondition(whiteKing,
rightWhiteRook, new Vector2(5, 0), new Vector2(6, 0)));
        blackKing.setCanCastleLeft(checkCastlingCondition(blackKing,
leftBlackRook, new Vector2(1, 7), new Vector2(2, 7), new Vector2(3, 7)));
        blackKing.setCanCastleRight(checkCastlingCondition(blackKing,
rightBlackRook, new Vector2(5, 7), new Vector2(6, 7)));

        //Turn finished - hand turn over to opponent.
        hud.changeTurn();
        hud.setGameMoves(PieceMovements.getGameMoves()); //update those moves
        if(turnFlag=='B') {
            hud.addGameMessage("AI moved " + piece.getPiece() + " to " +
boardColumns[x] + (y+1));
            if(whiteKing.isInCheck(board)) hud.addGameMessage("Your king is in
check!");

            turnFlag = 'W';
            return true;
        }
        if(turnFlag=='W') {
            turnFlag='B';
            return true;
        }
    }
    //If not, nothing happens.
    return false;
}
/**
 * Runs through the moveHistory string and moves all pieces on the board
accordingly.
 * @param moveHistory the string of moves for a game.
 */
public void setGame(String moveHistory) {
    char[] charArr = moveHistory.toCharArray();
    String nextMove = "";

    for(int i = 0; i < charArr.length; i++) {
        nextMove += charArr[i];
        if(nextMove.length()==4) {
            int previousX = Character.getNumericValue(nextMove.charAt(0));
            int previousY = Character.getNumericValue(nextMove.charAt(1));
            int newX = Character.getNumericValue(nextMove.charAt(2));
            int newY = Character.getNumericValue(nextMove.charAt(3));
            char promotion = '-';

            try{
                if(charArr[i+1] == 'Q' || charArr[i+1] == 'R' || charArr[i+1]
== 'B' || charArr[i+1] == 'K') {
                    promotion = charArr[i+1];
                    i++;
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                //The last move made was not a promotion - do nothing.
            }

            checkAndMove(board[previousY][previousX], newX, newY, promotion);
            nextMove = "";
        }
    }
}
/**

```

```

    * This method checks whether a castling move can be performed given the King
    and Rook involved and the positions in-between
    * them. In order for this method to return true, the following conditions must
    be met:
    * - The King involved in the move cannot be in check (The Rook involved can
    however be in check before a castling move is made).
    * - Both the King and Rook involved in the move should not have previously
    moved during the game.
    * - There can be no pieces in between the Rook and King involved in the
    castling move.
    * - The positions/tiles between the Rook and King cannot be checked by any
    opponent piece (i.e. You cannot castle through checked positions).
    * - The King cannot move into a checked position as part of the castling move.
    */
    private boolean checkCastlingCondition(King king, Rook rook, Vector2...
inBetweenPositions) { //returns whether or not this you can castle here

        //check if rook is alive
        if(board[(int)rook.getPosition().y][(int)rook.getPosition().x] == null)
return false;
        ChessPiece b1 =
board[(int)inBetweenPositions[0].y][(int)inBetweenPositions[0].x];
        ChessPiece c1 =
board[(int)inBetweenPositions[1].y][(int)inBetweenPositions[1].x];
        ChessPiece d1 = null;
        try {
            d1 = board[(int)inBetweenPositions[2].y][(int)inBetweenPositions[2].x];
        } catch(IndexOutOfBoundsException e) {}

        if(!king.isInCheck(board)) { //King isn't in check.
            if(!king.hasMoved() && !rook.hasMoved() ) { //King and Rook haven't
moved.
                if(b1 == null && c1 == null && d1 == null) { //Spaces between rook
and king are empty, if there are only two spaces, d1 will still be null.
                    for (ChessPiece[] row : board){
                        for (ChessPiece piece : row) {
                            if (piece != null &&
piece.getPiece().toString().charAt(0) != king.getPiece().toString().charAt(0)) {
//Opposite colour to King involved.
                                try {
                                    if(piece.getAllPotentialMoves().contains(inBetweenPositions[2])) {
                                        //Position 3 exists and is being checked.
                                        return false;
                                    }
                                    //Position 3 exists and isn't being checked.
                                } catch(IndexOutOfBoundsException e) {}
                                //Position 3 either exists and isn't being checked
OR doesn't exist.

                                    if
(piece.getAllPotentialMoves().contains(inBetweenPositions[0]) ||
piece.getAllPotentialMoves().contains(inBetweenPositions[1]) ) {
                                        return false; //Cannot castle.
                                    }
                                }
                            }
                        }
                    }
                    return true; //Can castle.
                }
            }
        }
        return false; //Cannot castle.
    }
}

```

```

/**
 * The main draw method of the game, renders all tiles and piece textures.
 */
private void draw() {
    //2D board
    for(int i = 0; i < board.length; i++) {
        for(int k = 0; k < board[i].length; k++) {

            if( (i%2==0 && k%2==0) || (i%2==1 && k%2==1) ) {
                sb.draw(tiles[0], tileSize*k, tileSize*i, tileSize, tileSize);
//lighter
            } else {
                sb.draw(tiles[1], tileSize*k, tileSize*i, tileSize, tileSize);
//darker
            }

            //Hover tile.
            if(k == (int)hoverTile.x && i == (int)hoverTile.y ) {
                sb.draw(tiles[2], tileSize*k, tileSize*i, tileSize, tileSize);
            }

            //Selection tiles.
            for(Vector2 tilePos : tileSelectionPositions) {
                if(k == (int)tilePos.x && i == (int)tilePos.y) {
                    sb.draw(tiles[3], tileSize*k, tileSize*i, tileSize,
tileSize);
                }
            }

        }

        //Area around board.
        for(int i = 0; i < board.length; i++) { //Rows.
            sb.draw(tiles[4], -tileSize/2, tileSize*i, tileSize/2, tileSize);
            sb.draw(tiles[4], tileSize*8, tileSize*i, tileSize/2, tileSize);
        }
        for(int k = 0; k < board[0].length; k++) { //Columns.
            sb.draw(tiles[4], tileSize*k, -tileSize/2, tileSize, tileSize/2);
            sb.draw(tiles[4], tileSize*k, tileSize*8, tileSize, tileSize/2);
        }
        sb.draw(tiles[4], (-tileSize/2), (-tileSize/2), tileSize/2, tileSize/2);
        sb.draw(tiles[4], (-tileSize/2), tileSize*8, tileSize/2, tileSize/2);
        sb.draw(tiles[4], tileSize*8, tileSize*8, tileSize/2, tileSize/2);
        sb.draw(tiles[4], tileSize*8, (-tileSize/2), tileSize/2, tileSize/2);

        //Pieces.
        for(ChessPiece[] row : board) {
            for(ChessPiece item : row) {
                if(item != null) { //Only convert to pixel positions here (*tile
size).
                    sb.draw(item.getPieceTexture(), item.getPosition().x *
tileSize, item.getPosition().y * tileSize,
                    tileSize, tileSize);
                }
            }
        }
    }
}

public boolean renderBoard() {

    //Update.

    if(!gameOver){
        gameOver = update();
        if(gameOver) gameInput.setGameOver(); //Stop dealing with input.
    }

    //Draw;

```

```

        draw();

        return gameOver;
    }
    private boolean isInCheckMate(char checkFor) {
        //Loop through every piece of opponent and see if all result in friendly
king checked.

        for(ChessPiece[] row : board) {
            for(ChessPiece piece : row) {
                if(piece != null &&
piece.getPiece().toString().charAt(0)==checkFor) { //Look at every enemy piece.
                    for(Vector2 move : piece.getAllPotentialMoves()) { //Go through
all it's moves.
                        PieceMovements.move(board, piece, (int)move.x,
(int)move.y);
                    if(!piece.getFriendlyKing(board).isInCheck(board)) { //If
any one saves the king.
                        PieceMovements.undo();
                        return false; //Then it is not a checkmate.
                    }
                    PieceMovements.undo();
                }
            }
        }

        return true;
    }
}
/**
 * Used for InputMultiplexer in GameScreen
 * @return gameInput - the InputProcessor used for this game.
 */
public GameInput getInput() {
    return gameInput;
}
@Override
public void dispose() {
    for(Texture texture : tiles) {
        texture.dispose();
    }
    for(ChessPiece[] row : board) { //Dispose whatever is left on the board.
        for(ChessPiece item : row) {
            if(item != null)
                item.dispose();
        }
    }
}
}
}

```

GameScreen

```

package com.dylanwalsh.chessai.screens;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.InputMultiplexer;
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.dylanwalsh.chessai.database.DBConnection;

/**
 * The class of the game that deals with the board and hud. This is the main screen
of the game. It implements the Screen interface
 * from the LibGDX libraries.

```

```

*/
public class GameScreen implements Screen {
    public static final float VIEWPORT_WIDTH = 370;
    private Board board;
    private SpriteBatch sb;
    private OrthographicCamera gameCam;
    private HUD hud;
    private InputMultiplexer input;
    private boolean changingGame = false;
    private String moveHistory = ""; //Move history for new loaded game.

    public GameScreen() {
        sb = new SpriteBatch();
        gameCam = new OrthographicCamera();

        DBConnection.initialize();

        hud = new HUD(this);
        board = new Board(sb, gameCam, hud);

        input = new InputMultiplexer();
        input.addProcessor(hud.getInput());
        input.addProcessor(board.getInput());
        Gdx.input.setInputProcessor(input);
    }
    public void setGame(String moveHistory) {
        changingGame = true;
        this.moveHistory = moveHistory;
    }
    @Override
    public void show() {}
    @Override
    public void render(float delta) {

        Gdx.gl.glClearColor( .25f, .25f, .25f, 1 );
        Gdx.gl.glClear( GL20.GL_COLOR_BUFFER_BIT | GL20.GL_DEPTH_BUFFER_BIT );

        sb.setProjectionMatrix(gameCam.combined);
        sb.begin();
        if(!changingGame) {
            board.renderBoard();
        } else { //If the game is being changed, stop calling renderBoard() or
Board class.
            board.dispose();
            input.removeProcessor(1);
            board = new Board(sb, gameCam, hud);
            input.addProcessor(board.getInput());
            board.setGame(moveHistory);
            changingGame = false;
            moveHistory = "";
        }

        sb.end();
        hud.renderHUD();
    }
    @Override
    public void resize(int width, int height) {
        //Sets gameCam to orthographic projection centered at 10px right of the
right edge of the board, height/2.
        gameCam.setToOrtho(false, VIEWPORT_WIDTH,
(float)height*(VIEWPORT_WIDTH/(float)width));
        gameCam.position.set(160+10, 160/2, 0);
        gameCam.update();
    }
    @Override
    public void pause() {}
}

```

```

@Override
public void resume() {}
@Override
public void hide() {
    dispose();
}
@Override
public void dispose() {
    sb.dispose();
    board.dispose();
    hud.dispose();
}
}

```

HUD

```

package com.dylanwalsh.chessai.screens;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.scenes.scene2d.InputEvent;
import com.badlogic.gdx.scenes.scene2d.InputListener;
import com.badlogic.gdx.scenes.scene2d.Stage;
import com.badlogic.gdx.scenes.scene2d.ui.Image;
import com.badlogic.gdx.scenes.scene2d.ui.Label;
import com.badlogic.gdx.scenes.scene2d.ui.ScrollPane;
import com.badlogic.gdx.scenes.scene2d.ui.Skin;
import com.badlogic.gdx.scenes.scene2d.ui.Table;
import com.badlogic.gdx.scenes.scene2d.ui.TextButton;
import com.badlogic.gdx.scenes.scene2d.ui.TextField;
import com.badlogic.gdx.scenes.scene2d.utils.ClickListener;
import com.badlogic.gdx.utils.Disposable;
import com.badlogic.gdx.utils.Queue;
import com.dylanwalsh.chessai.database.DBConnection;
import com.dylanwalsh.chessai.database.GameData;
import com.dylanwalsh.chessai.entities.ChessPiece;

import java.util.ArrayList;
import java.util.Timer;
import java.util.TimerTask;

/**
 * This class defines the on screen HUD. It is made up of table, text field, text
 * button and labels widgets. It implements the
 * Disposable interface so that the Stage, as well as widgets on the stage, and the
 * skin used to style the widgets can be disposed.
 */
public class HUD implements Disposable{

    private Stage stage;
    private Skin skin;
    private boolean gameOver;
    private Table table;
    private Table pieceTable;
    private TextField usernameField;
    private TextButton recordGameButton;
    private Table pastGamesTable;
    private TextButton loadGameButton;
    private Label timePlayedLabel;
    private Label movesLabel;
    private Label gameMessage; //Latest game message.
    //This is the game hud's messageQueue. It is an implementation of a queue that
    //stored string items. The addMessage() method
    //will add a new String to the message queue to be displayed on screen.
    private Queue<String> messageQueue;
    private boolean playerTurn;
    private int time;

```

```

private int moves;
private int win; //Stores whether player won, lost or hasn't finished.
private String gameMoves; //A string of game moves, this is just a long
sequence of numbers stored as TEXT in the database.
private GameData loadedGame = null;

public HUD(final GameScreen gameScreen) {
    stage = new Stage();

    stage.getRoot().addCaptureListener(new InputListener() {
        public boolean touchDown (InputEvent event, float x, float y, int
pointer, int button) {
            if (!(event.getTarget() instanceof TextField))
stage.setKeyboardFocus(null);
            return false;
        }
    });

    skin = new Skin(Gdx.files.internal("hud/uiskin.json"));
    gameOver = false;

    table = new Table();

    table.setWidth((Gdx.graphics.getWidth()/2f)-80);
    table.setPosition((Gdx.graphics.getWidth()/2f)+40,
(Gdx.graphics.getHeight()/4f)*2.2f);

    timePlayedLabel = new Label("", skin, "default");
    movesLabel = new Label("Moves: 0", skin, "default");
    gameMessage = new Label("", skin, "default");
    messageQueue = new Queue<String>();
    pieceTable = new Table();
    usernameField = new TextField("", skin, "default");
    usernameField.setMessageText("Enter a Username!");

    //Set gameMoves to empty string in case a save before first move.
    gameMoves = "";

    recordGameButton = new TextButton("Record Game", skin, "default");
    //Definition of new anonymous inner class overriding clicked method to deal
with record game button being clicked.
    recordGameButton.addListener(new ClickListener() {
        private boolean clickedUsername = false;
        private boolean clickedLength = false;
        @Override
        public void clicked(InputEvent event, float x, float y) {
            //Save game data to database.

            //Only save brand new games or old games unfinished.
            if(loadedGame==null || loadedGame.getWin()==2) {
                if(loadedGame==null) { //brand new game
                    if(usernameField.getText().isEmpty()) {
                        if(!clickedUsername) {
                            addGameMessage("Enter a Username!");
                            clickedUsername = true;
                        }
                        return; //Make shore there is a username present.
                    }
                    if(usernameField.getText().length() > 20) {
                        if(!clickedLength) {
                            addGameMessage("Max Username Length is 20!");
                            clickedLength = true;
                        }
                        return; //Make shore the length of username is less
than or equal to 20.
                    }
                }
            }
            //If updating a game it wont matter what usernameField is.

```



```

        GameData newlySavedGame =
DBConnection.saveGame(usernameField.getText(),
        time, moves, gameMoves, gameOver?win:2,
loadedGame==null?-1:loadedGame.getId());
        addGameMessage(loadedGame==null?"Game Saved!":"Game Updated!");

        //Click load game.
        InputEvent buttonDown = new InputEvent();
        buttonDown.setType(InputEvent.Type.touchDown);
        InputEvent buttonUp = new InputEvent();
        buttonUp.setType(InputEvent.Type.touchUp);
        loadGameButton.fire(buttonDown);
        loadGameButton.fire(buttonUp);

        if(loadedGame == null)
            loadedGame = newlySavedGame;

        usernameField.setText("");
        usernameField.setDisabled(true);
    }
});

pastGamesTable = new Table();
pastGamesTable.setFillParent(true);
pastGamesTable.bottom();

final ScrollPane scrollPane = new ScrollPane(pastGamesTable, skin,
"default");
scrollPane.setHeight(65);
scrollPane.setWidth((Gdx.graphics.getWidth()/2f)-80);
scrollPane.setPosition((Gdx.graphics.getWidth()/2f)+40,
Gdx.graphics.getHeight()/14f);
scrollPane.setScrollingDisabled(true, false);
scrollPane.setFadeScrollBars(false);
scrollPane.setVisible(false);

loadGameButton = new TextButton("Load Game", skin, "default");
//Definition of new anonymous inner class overriding clicked method to deal
with load game button being clicked.
loadGameButton.addListener(new ClickListener() {
    @Override
    public void clicked(InputEvent event, float x, float y) {

        ArrayList<GameData> games = DBConnection.getPastGames();
        pastGamesTable.clearChildren();
        scrollPane.setVisible(true);

        //name : timestamp : score : win/loss/unfinished
        for(GameData g : games) {
            final GameData game = g; //Declared final so that click
listener inner class can access.
            String gameString =
                game.getName() + " : " +
                game.getTimeStamp().toString() + " : score -> " +
                game.getScore() + " : " +

(game.getWin()==0?"Loss):(game.getWin()==1?"Win":"Unfinished"));
            Label l = new Label(gameString, skin, "default");
            l.setFontScale(.9f, .9f);
            pastGamesTable.add(l).expandX();
            if( ! (games.indexOf(game) == (games.size()-1)) )
pastGamesTable.row();

        //Definition of new inner class overriding clicked method to

```

```

deal with GameData instance button being clicked.
    l.addListener(new ClickListener() {
        @Override
        public void clicked(InputEvent event, float x, float y) {
            loadedGame = game;
            pieceTable.clearChildren();
            time = loadedGame.getTimeSurvived();
            String s = Integer.toString(time);
            s = s.replaceAll("\\B(?=(?:..)+$)", ":");
            timePlayedLabel.setText("Time: "+s);
            messageQueue.clear();
            moves = 0;
            usernameField.setText("");
            usernameField.setDisabled(true);
            gameOver = game.getWin()==2?false:true;

            //Load the game.
            gameScreen.setGame(loadedGame.getMoveHistory());
        }
    });
}
});

table.add(timePlayedLabel).expandX().padBottom(5);
table.row();
table.add(movesLabel).expandX().padBottom(20);
table.row();
table.add(gameMessage).expandX().padBottom(20);
table.row();
table.add(pieceTable).expandX().padBottom(20);
table.row();
table.add(usernameField).expandX().padBottom(5);
table.row();
table.add(recordGameButton).expandX().padBottom(5);
table.row();
table.add(loadGameButton).expandX();

stage.addActor(table);
stage.addActor(scrollPane);

playerTurn = true;
time = 0;
new Timer().scheduleAtFixedRate(new TimerTask() {
    @Override
    public void run() {
        if(playerTurn && !gameOver) {
            if(time % 100 == 59) {time += 41;}
            else {time++;}
            String s = Integer.toString(time);
            s = s.replaceAll("\\B(?=(?:..)+$)", ":");
            timePlayedLabel.setText("Time: "+s);
        }
    }
}, 0, 1000);
}
public void changeTurn() {
    playerTurn = !playerTurn;

    //If false, player handed to AI, therefore player made a move.
    if(playerTurn == false) {
        moves++;
        movesLabel.setText("Moves: " + Integer.toString(moves));
    }
}
}

```

```

public void setGameOver(int win) {
    gameOver = true;
    this.win = win;
}
public void setGameMoves(String gameMoves) {
    this.gameMoves = gameMoves;
}
public void addTakenPiece(CheessPiece piece) {
    Texture takenPieceTexture;
    switch(piece.getPiece()) {
        case BBISHOP:
            takenPieceTexture = new Texture("chess_pieces/bbishop.png");
            break;
        case WBISHOP:
            takenPieceTexture = new Texture("chess_pieces/wbishop.png");
            break;
        case BKING:
            takenPieceTexture = new Texture("chess_pieces/bking.png");
            break;
        case WKING:
            takenPieceTexture = new Texture("chess_pieces/wking.png");
            break;
        case BKNIGHT:
            takenPieceTexture = new Texture("chess_pieces/bknight.png");
            break;
        case WKNIGHT:
            takenPieceTexture = new Texture("chess_pieces/wknight.png");
            break;
        case BQUEEN:
            takenPieceTexture = new Texture("chess_pieces/bqueen.png");
            break;
        case WQUEEN:
            takenPieceTexture = new Texture("chess_pieces/wqueen.png");
            break;
        case BROOK:
            takenPieceTexture = new Texture("chess_pieces/brook.png");
            break;
        case WROOK:
            takenPieceTexture = new Texture("chess_pieces/wrook.png");
            break;
        case BPAWN:
            takenPieceTexture = new Texture("chess_pieces/bpawn.png");
            break;
        case WPAWN:
            takenPieceTexture = new Texture("chess_pieces/wpawn.png");
            break;
        default:
            takenPieceTexture = new Texture("chess_pieces/wpawn.png");
            break;
    }
    Image takenPiece = new Image(takenPieceTexture);
    pieceTable.add(takenPiece).width(40).height(40);
    if((pieceTable.getChildren().size)%8==0) pieceTable.row();
}

public void addGameMessage(String message) {
    messageQueue.addLast(message);
    if(messageQueue.size == 4) messageQueue.removeFirst();
    String msg = "";
    for(String s : messageQueue) {
        msg += s+"\n";
    }
    msg = msg.substring(0, msg.length()-1);
    gameMessage.setText(msg);
}

public void renderHUD() {
    stage.act();
    stage.draw();
}
}
/**

```

```

    * Used for InputMultiplexer in GameScreen.
    * @return The stage, which itself is an InputProcessor.
    */
    public Stage getInput() {
        return stage;
    }
    @Override
    public void dispose() {
        stage.dispose();
        skin.dispose();
    }
}

```

com.dylanwalsh.chessai.util

PieceMovement

```

package com.dylanwalsh.chessai.util;

import com.dylanwalsh.chessai.entities.ChessPiece;

/**
 * This class represents a single piece movement, it keeps track of information
 * associated with moving a piece. This includes the
 * piece to move, where to move it to, where it is moving from, if there is a piece
 * in the position it is moving to, whether this
 * movement is part of a castling move (which would mean that it is either a rook
 * or king).
 */
public class PieceMovement {
    private ChessPiece piece;
    private int fromX;
    private int fromY;
    private int toX;
    private int toY;
    //If this is a rook or king castling.
    private boolean castle;
    //If there is a piece at that position, record it for later use. Will be null
    if no piece is there.
    private ChessPiece pieceAtPosition;

    public PieceMovement(ChessPiece piece, int toX, int toY, ChessPiece
    pieceAtPosition, boolean castle) {
        this.piece = piece;
        this.fromX = (int)piece.getPosition().x;
        this.fromY = (int)piece.getPosition().y;
        this.toX = toX;
        this.toY = toY;
        this.pieceAtPosition = pieceAtPosition;
        this.castle = castle;
    }
    public boolean isCastle() {
        return castle;
    }
    public ChessPiece getPiece() { return piece; }
    public ChessPiece getPieceAtPosition() { return pieceAtPosition; }
    public int getFromX() { return fromX; }
    public int getFromY() {
        return fromY;
    }
    public int getToX() {
        return toX;
    }
    public int getToY() {
        return toY;
    }
}

```

PieceMovements

```
package com.dylanwalsh.chessai.util;

import com.badlogic.gdx.math.Vector2;
import com.dylanwalsh.chessai.entities.ChessPiece;
import com.dylanwalsh.chessai.entities.chesspieces.King;
import com.dylanwalsh.chessai.entities.chesspieces.Pawn;
import com.dylanwalsh.chessai.entities.chesspieces.Rook;
import com.dylanwalsh.chessai.input.GameInput;

import java.util.Stack;

/**
 * A static class used in GameAI and ChessPiece that performs and undoes piece
 * movements. It makes use of a stack to track the
 * history of moved pieces.
 */
public class PieceMovements {
    //This is the piece move stack. A move is pushed onto the move stack when a new
    //piece is moved. The piece is removed from the
    //stack when a move is undone. Piece movements that are not undone are left in
    //the stack.
    private static Stack<PieceMovement> moveStack = new Stack<PieceMovement>();
    private static String moveString = "";
    private static ChessPiece[][] chessBoard;

    public static ChessPiece move(ChessPiece[][] board, ChessPiece piece, int posX,
int posY) {
        chessBoard = board;
        PieceMovement p = null;

        if(piece.getPiece() == ChessPiece.Pieces.BKING || piece.getPiece() ==
ChessPiece.Pieces.WKING) {
            King castleKing = (King)piece;
            Rook castleRook = null;
            if(castleKing.canCastleLeft() && posX ==
castleKing.getCastleLeftPosition().x && posY ==
castleKing.getCastleLeftPosition().y)
                castleRook = castleKing.getCastleLeftRook();
            else if(castleKing.canCastleRight() && posX ==
castleKing.getCastleRightPosition().x && posY ==
castleKing.getCastleRightPosition().y)
                castleRook = castleKing.getCastleRightRook();

            if(castleKing != null && castleRook != null) { //Castling move.

                //Push king and then rook onto the move stack (has to be in this
                //order as king is found in undo method by popping
                //off the stack the next movement after a castling rook).
                p = moveStack.push(new PieceMovement(piece, posX, posY,
board[posY][posX], true));
                moveStack.push(new PieceMovement(castleRook,
(int)castleRook.getCastlePosition().x, (int)castleRook.getCastlePosition().y,
board[posY][posX], true));

                //Set both positions to null first.
board[(int)castleRook.getPosition().y][(int)castleRook.getPosition().x] = null;
board[(int)castleKing.getPosition().y][(int)castleKing.getPosition().x] = null;

                //Update rook position.
board[(int)castleRook.getCastlePosition().y][(int)castleRook.getCastlePosition().x]
= castleRook;
                castleRook.setPosition(castleRook.getCastlePosition());
            }
        }
    }
}
```

```

        //Update king position.
        board[posY][posX] = castleKing;
        castleKing.setPosition(new Vector2(posX, posY));

    } else if(castleRook == null) { //Regular king movement.
        //Here the king movement is pushed onto the move stack.
        p = moveStack.push(new PieceMovement(piece, posX, posY,
board[posY][posX], false));

        board[p.getFromY()][p.getFromX()] = null; //Set previous position
to null.

        board[posY][posX] = piece; //Update new position on board array.
        piece.setPosition(new Vector2(posX, posY)); //Set piece position
vector.
    }

    } else {
        //Push a regular piece movement onto the stack.
        p = moveStack.push(new PieceMovement(piece, posX, posY,
board[posY][posX], false));

        board[p.getFromY()][p.getFromX()] = null; //Set previous position to
null

        board[posY][posX] = piece; //Update new position on board array.
        piece.setPosition(new Vector2(posX, posY)); //Set piece position
vector.
    }

    //Generate new positions for every piece.
    for(ChessPiece[] row : board) {
        for(ChessPiece pi : row) {
            if(pi != null) pi.generatePositions(board);
        }
    }

    return p.getPieceAtPosition(); //So it can get disposed if it was taken.
}
/**
 * Undoes the previous movement. This method calls generatePositions on every
piece after it undoes the previous movement.
 */
public static void undo() {

    if(!moveStack.isEmpty()) {
        //Pop the next movement off the move stack.
        PieceMovement p = moveStack.pop();

        if( (p.getPiece().getPiece() == ChessPiece.Pieces.WROOK ||
p.getPiece().getPiece() == ChessPiece.Pieces.BROOK) && p.isCastle() == true) {

            //The PieceMovement object after a castling rook movement has been
popped off the stack should be the movement of the
            //king in the castling move.
            PieceMovement pKing = moveStack.pop();

            chessBoard[pKing.getFromY()][pKing.getFromX()] = pKing.getPiece();
//Set previous position to piece.
            chessBoard[pKing.getToY()][pKing.getToX()] =
pKing.getPieceAtPosition(); //Will be null if no piece was at that position.
            pKing.getPiece().setPosition(new Vector2(pKing.getFromX(),
pKing.getFromY())); //Update piece position vector.
            //Rook will get reset below.
        }

        //Undo movement.

        chessBoard[p.getFromY()][p.getFromX()] = p.getPiece(); //Set previous

```

```

position to piece.
    chessBoard[p.getToY()][p.getToX()] = p.getPieceAtPosition(); //Will be
null if no piece was at that position.
    p.getPiece().setPosition(new Vector2(p.getFromX(), p.getFromY()));
//Update piece position vector.

    //Finish by generating all new positions for every piece.
    for(ChessPiece[] row : chessBoard) {
        for(ChessPiece piece : row) {
            if(piece != null) piece.generatePositions(chessBoard);
        }
    }
}

/**
 * This method returns a combination of all moves made throughout the course of
the game so far in the form of a String.
 * The format for a move would be:
 * previousx previousy newx newy (promotePawn character)
 *
 * So a string that looks like 3545 would mean the piece at 6D moved to 6E
 * A string that looks like 6667Q would mean the white pawn at G7 moved to G8
and was promoted to a Queen.
 * @return a string representing the game move history so far.
 */
public static String getGameMoves() {
    return moveString;
}
public static void resetMoves() { moveStack.empty(); moveString=""; }

/**
 * Records the last game added to the move stack in the moveString.
 */
public static void recordLastMove() {
    PieceMovement m = moveStack.lastElement();
    moveString += Integer.toString(m.getFromX()) +
Integer.toString(m.getFromY()) + Integer.toString(m.getToX()) +
Integer.toString(m.getToY());
}
public static void promoteWhitePawn(Pawn pawn, char promotePawn, GameInput in)
{
    char promotedPawn = in.promotePawn(pawn, promotePawn);
    moveString += promotedPawn;
}
public static void promoteBlackPawn(Pawn pawn, char promotePawn) {
    char promotedPawn = pawn.promote(chessBoard, promotePawn);
    moveString += promotedPawn;
}
}

```

GameClass

```

package com.dylanwalsh.chessai;

import com.badlogic.gdx.Game;
import com.dylanwalsh.chessai.screens.GameScreen;

public class GameClass extends Game {
    @Override
    public void create () {
        setScreen(new GameScreen());
    }
    @Override
    public void render () {
        super.render();
    }
}

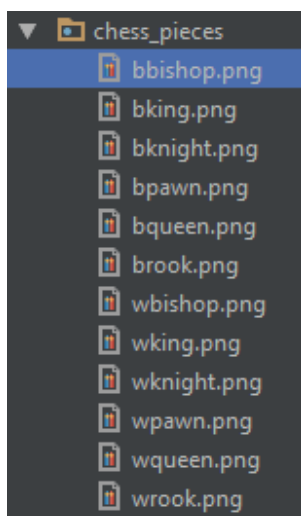
```

```
}
@Override
public void dispose () {
    super.dispose();
}
}
```

Assets

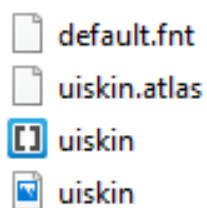
chess_pieces

This folder just contains all the images (png files) for each chess piece for white and black.



Hud

The hud folder contains all the files I need to create my skin in libgdx. The skin in libgdx is what you use to style widget. There are four files in this folder; 1 font file and 3 files for the uiskin.








default.fnt is the font file.

uiskin.atlas, uiskin.json and uiskin.png contain information about styling the ui widgets.

tiles

The tiles folder contains 5 images. The hover to is draw when the mouse is hovered over a particular tile. The move tile is drawn when a piece is selected. 1, 2 and 3 are the tiles used to render the game board.

-  1
-  2
-  3
-  hover
-  move

System Testing

Test Tables

Piece Functionality Tests

Here I conduct tests for the functionality of each piece. This involves checking that the program follows the official rules of chess (piece's can only make legal moves, pawn promotion works properly, all castling conditions must be met before a castling move can be performed, etc.).

Test Number	Description	Input Data	Expected Outcome/Output Data	Actual Outcome/Output Data	Reference
1	Successfully render all tiles on the board.	ChessPiece[][] board – 2D board array representing the game board	8x8 grid of tiles rendered on screen	As expected	1.1
2	Successfully place white pawn on board (add Pawn object to 2D board array).	Index within board to add piece (x: 0, y: 1) - integers	White Pawn object added to board array at position (0, 1)	As expected – board contains 1 initialised ChessPiece object	
3	Successfully place all other white and black pieces on the board.	Appropriate index within board to place each piece - integers	All ChessPiece objects added to board array at appropriate positions	As expected – board contains 32 initialised ChessPiece objects	
4	Successfully render chess pieces on the board.	ChessPiece[][] board – 2D board array representing the game board	Contents of board array should be rendered on screen	As expected	1.2
5	Check outcome of placing a	Index within board to add piece, outside	java.lang.ArrayIndexOutOfBoundsException thrown	As expected	1.3

	ChessPiece object outside the range of the board.	board (x: -1, y: -1) - integers			
<p>The structure I followed when testing which position each piece can move to is as follows:</p> <ol style="list-style-type: none"> (1) – Enter a position within a piece’s official move set. (2) – Enter a position outside a piece’s official move set. (3) – Enter a position within a piece’s official move set where the position is already occupied by another piece. (4) – Enter a position within a piece’s official move set where the position is restricted by another piece. (5) – Special cases 					
6	Check move set of Pawn – (1)	Pawn position is (x: 0, y: 1), data entered is (x:0, y:2) - integers	Pawn moved forward one place	As expected	1.4
7	Check move set of Pawn – (2)	Pawn position is (x: 0, y: 1), data entered is (x: 1, y: 1) - integers	No changes in the state of the board	As expected	
8	Check move set of Pawn – (3) – Not attacking move	Pawn position is (x: 3, y: 3), data entered is (x: 3, y: 4) - integers	No changes in the state of the board	As expected	1.5
9	Check move set of Pawn – (5) – First move forward two positions	Pawn position is (x: 0, y: 1), data entered is (x: 0, y: 3) – integers	Pawn moved forward two places	As expected	1.6
10	Check move set of Queen – (4)	Queen position is (x: 2, y: 2), data entered is (x: 7, y: 7) – integers	No changes in the state of the board	As expected	1.7
11	Check move set of Knight – (1)	Knight position is (x: 1, y: 0), data entered is (x: 2, y: 2) - integers	Knight moved up 2, right 1	As expected	1.8
12	Check move set of Rook – (3) –	Rook position is (x: 3, y: 4), data entered	Rook took Bishop ChessPiece at position on the board	As expected	1.9

	Attacking move	is (x: 6, y: 4) - integers			
13	Check move set of Pawn – (5) – Pawn promotion	Pawn position is (x: 6, y: 6), data entered is (x: 7, y: 7) – integers. Also entered '1' for Rook (see screenshot)	Pawn promoted to Rook	As expected	2.0
14	Special move involving King and Rook – Castling. All conditions met.	King position is (x: 4, y: 0), data entered is (x: 6, y: 0).	No castle move was performed.	Castle movement should be performed. NullPointerException occurred, issue was fixed and then castle move was performed properly.	2.1
15	Special move involving King and Rook – Castling. Cannot castle through checked positions and king cannot move into checked position.	King castle to x: 2, y: 0.	Castle move is not performed.	As expected.	2.2
16	Special move involving King and Rook – Castling. King cannot be in check.	King castle to x: 2, y: 0.	Castle move is not performed.	As expected.	2.3
17	Special move involving King and	Moved King 1 place and then moved king back and	Castle move is performed.	Expected no move to be performed. I did not set	

	Rook – Castling. Both the king and rook involved should not have moved during the game.	attempted a castle move. Repeated for rook. King castle to x: 2, y: 0		the moved flag for king or rook. Castle move is now not performed.	
18	Special move involving King and Rook – Castling. There can be no pieces between the rook and king.	King castle to x: 2, y: 0	Castle move not performed.	As expected.	2.4

Other Tests

Test Number	Description	Input Data	Expected Outcome/Output Data	Actual Outcome/Output Data	Reference
<p>In order to test the minimax algorithm along with its alpha beta improvement, I chose to record the number of positions evaluated as well as the execution time of both methods in order to compare the two. This way, I could see that the alpha-beta improvement executed faster and evaluated less nodes (meaning there was an improvement). For the minimax algorithm, I also recorded the evaluation made at the leaf nodes by the algorithm for one move at search depths 2 and 3. This way I could see that the algorithm was recursive, and that it was evaluating a greater number of nodes with a greater depth.</p>					
19	Minimax algorithm – more nodes are evaluated at a greater depth (for one move).	Moved a single piece on the board for search depth 2, repeated for search depth 3.	More positions were evaluated at a search depth of 3.	Expected outcome.	2.5
20	Minimax algorithm – Execution time improvement	Move a single piece on the board. Checked evaluations for	Execution time on average would be quicker for alpha-beta than minimax.	There was not a significant improvement on average. After fixing an issue	2.6

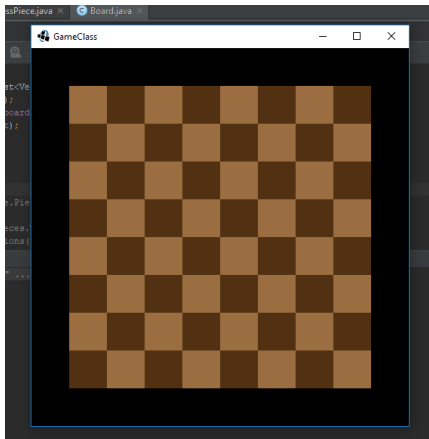
	nt for alpha-beta.	the minimax and alpha-beta.		inside the alpha-beta method, I repeated the test and saw there was a significant improvement in the execution time for alpha-beta.	
21	Game messages are displayed properly in a queue on screen.	Sequence of black moves: Black Bishop to G4 -> Black Bishop to D1 -> Black Pawn to E6.	“ AI moved BBISHOP to G4 AI moved BBISHOP to D1 AI moved BPAWN to E6 “	“ AI moved BPAWN to E6 AI moved BBISHOP to D1 AI moved BBISHOP to G4 “ This was in reverse order, I changed to print the string in the opposite order and then repeated the test. This produced the expected output	2.7
22	The time label increments only on the players turn.	Moved piece but made shore the turnFlag in Board did not get changed.	The time would stop during the black player’s turn.	As expected.	
23	The moves label increments only after the player has finished their turn.	Moved piece.	After a black piece is moved in response to the players first move, the move counter should be 1, not 2.	As expected.	
24	User is prompted about move made by AI to correct location.	Moved piece to 5, 3 to trigger black piece movement response.	Black knight moved to 0, 5 on board. Output should be ‘AI moved BKNIGHT to C6’.	‘AI moved BKNIGHT to D6’. Indexed boardColumns array incorrectly. IndexOutOfBoundsException was thrown for black piece movement to final column.	2.8

				Fixed issue, repeated test and got expected output.	
25	User is prompted when their king is in check.	Moved king to 4, 3. Black knight moved to 5, 5 to put king in check.	'Your king is in check!' added to the bottom of the message queue.	As expected.	2.9
26	User is prompted when the opponent has been checkmate.	Moved Queen to 7, 7	'White has Won!'	As expected	3.0
27	User is prompted when they have been check mate.	Moved King to 6, 4	'Black has Won!'	As expected	3.1
27	User is prompted of the next pawn promotion.	Pressed 'R' key for pawn promotion to rook.	'Next pawn promotion: Rook'	As expected	3.2
26	Taken pieces are added to the table in the hud.	Moved pawn to take black pawn.	Black pawn will be added to piece table.	A new black pawn texture was added to the table as expected.	3.3
27	Attempt to move enemy piece.	Select enemy bishop and move to one of it's legal moves.	Nothing happens	Piece was not moved but the turn was changed. Fixed error by making turnFlag change only if state of the board changes.	
28	Record Game button pressed when server not running.	Pressed Record game button.	SQLException should be caught and handled.	As expected, program output 'SQLException thrown, server may not be running!'.	3.4
29	Load Game button pressed when	Pressed Load game button.	SQLException should be caught and handles.	As expected, program output 'SQLException thrown, server	3.4

35	Textfield can only be modified when new game is being saved.	Attempt to modify textfield on old loaded game.	Textfield should be disabled.	Could not enter data into the username field.	
36	Type SQL statement into textfield to check SQL injection cannot occur.	Entered into textfield 'SELECT * FROM users;'	PreparedStatement should prevent SQL injections.	No SQLException thrown, username was stored properly.	4.0
<p>Query 1 - <code>INSERT INTO users(name) VALUES(?);</code></p> <p>Query 2 - <code>INSERT INTO games(time_stamp, time_survived, score, move_history, user_id, win_loss) VALUES((SELECT CURDATE()), ?, ?, ?, (SELECT id FROM users WHERE name = ?), ?);</code></p> <p>Query 3 - <code>UPDATE games SET time_stamp=(SELECT CURDATE()), time_survived=?, score=?, move_history=?, win_loss=? WHERE id=?;</code></p>					
37	Check query 1 inserts row into users table properly.	Entered name into text field and pressed record button.	Table should contain new row.	As expected.	4.1
38	Check query 2 inserts row into games table properly.	Saved a new game with record game button.	Table should contain new row.	As expected.	4.2
39	Check query 3 properly updates games table.	Load old game and then re-save the game after a few moves.	Table updates row with specified id.	All existing rows were updated. This was because I forgot to add 'WHERE ID=?' into the query. Fixed issue and re-run, table was now updating properly.	4.3

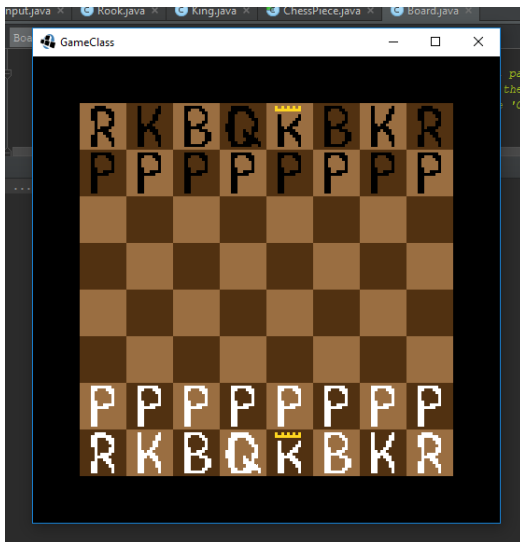
Screenshots

Screenshot 1.1



The rendered chessboard tiles.

Screenshot 1.2



All rendered pieces on the board.

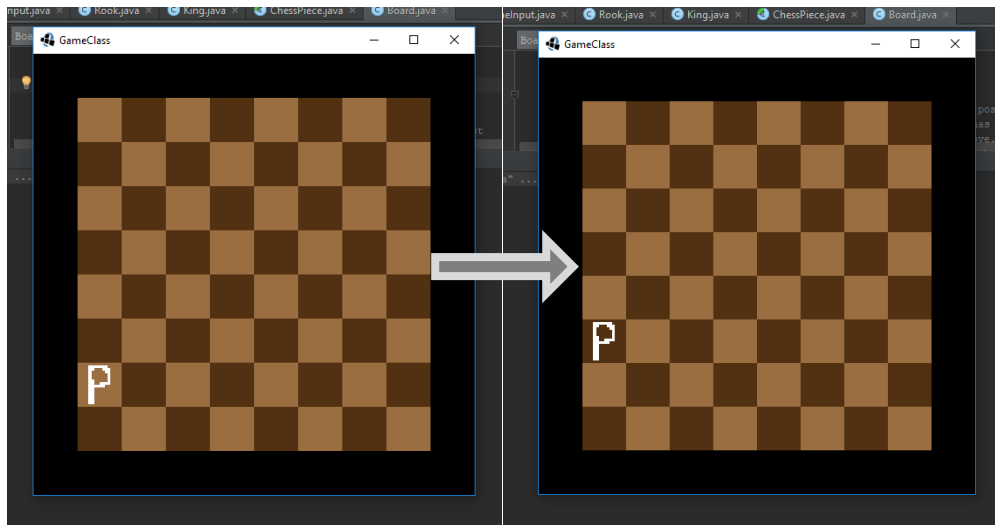
Screenshot 1.3

```
Exception in thread "LWJGL Application" java.lang.ArrayIndexOutOfBoundsException: -1
    at com.dylanwalsh.chessai.entities.Board.<init>(Board.java:67)
    at com.dylanwalsh.chessai.screens.GameScreen.<init>(GameScreen.java:26)
    at com.dylanwalsh.chessai.GameClass.create(GameClass.java:17)
    at com.badlogic.gdx.backends.lwjgl.LwjglApplication.mainLoop(LwjglApplication.java:149)
    at com.badlogic.gdx.backends.lwjgl.LwjglApplication$1.run(LwjglApplication.java:126)

Process finished with exit code 0
```

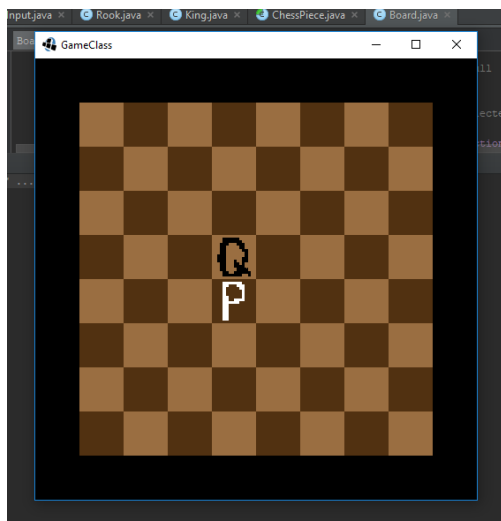
java.lang.ArrayIndexOutOfBoundsException thrown when ChessPiece placed outside range of board.

Screenshot 1.4

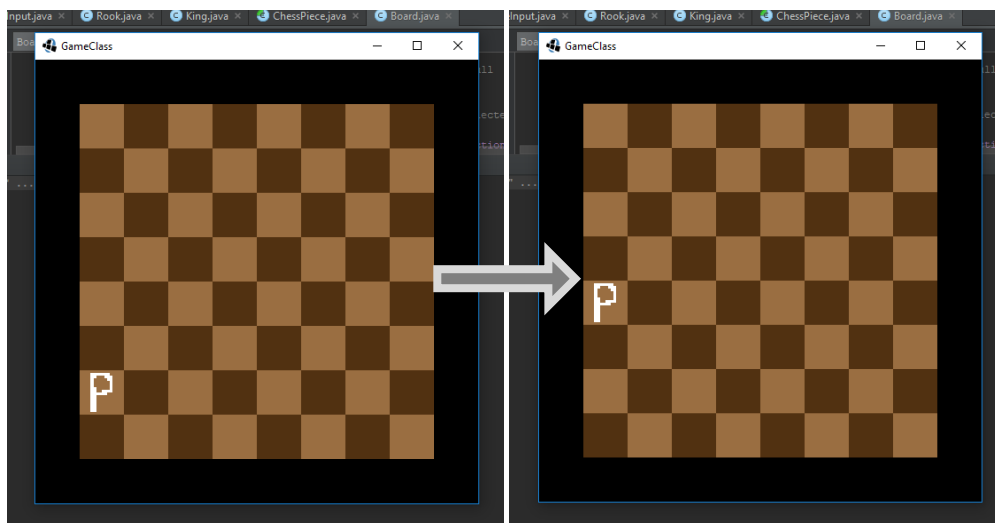


Pawn moved forward one space.

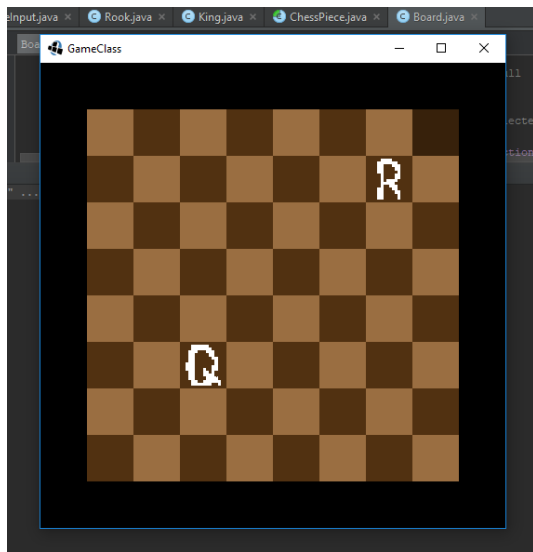
Screenshot 1.5



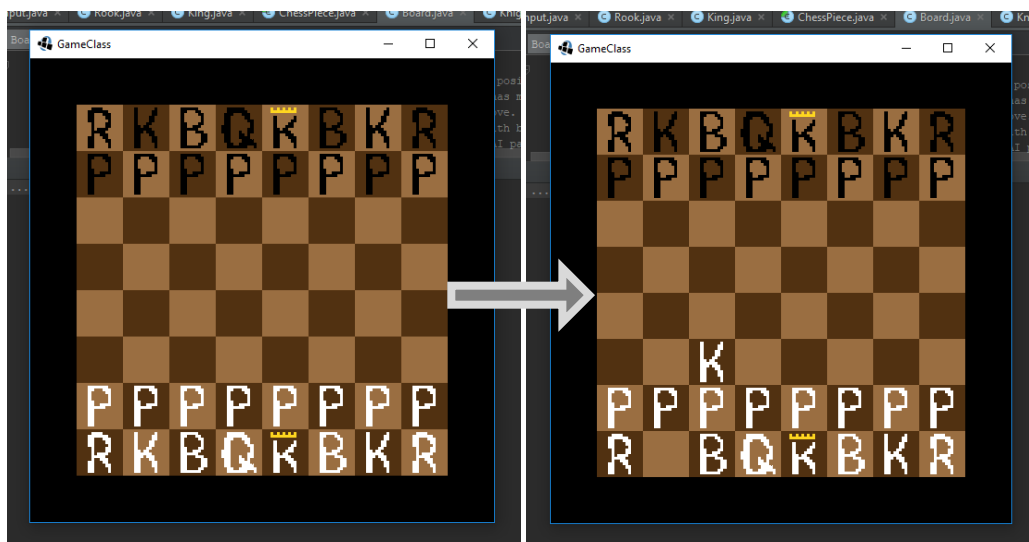
Screenshot 1.6



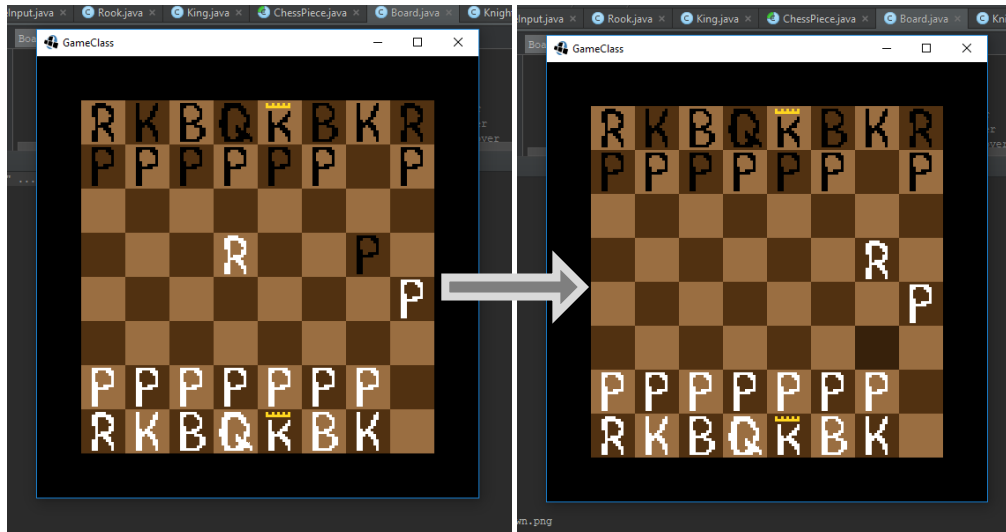
Screenshot 1.7



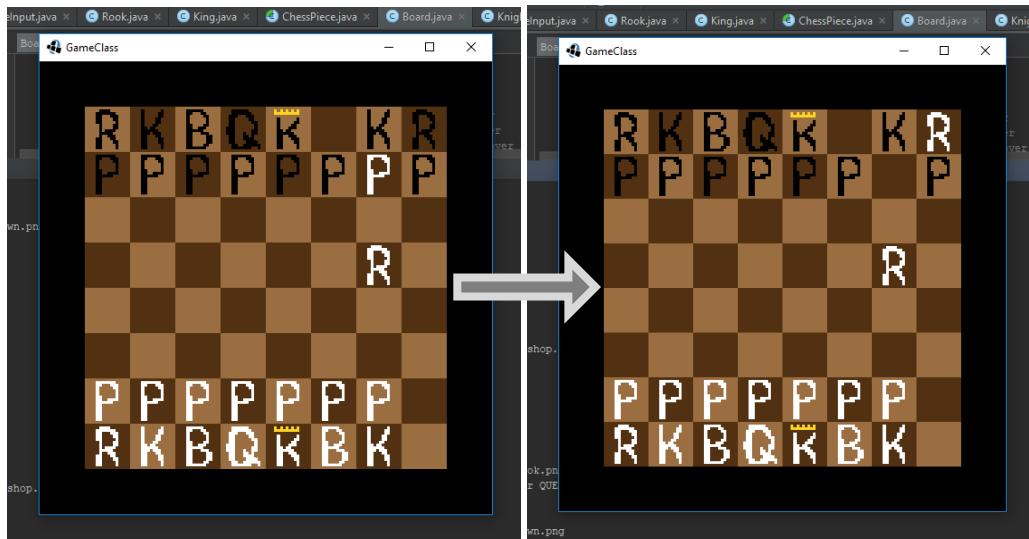
Screenshot 1.8



Screenshot 1.9

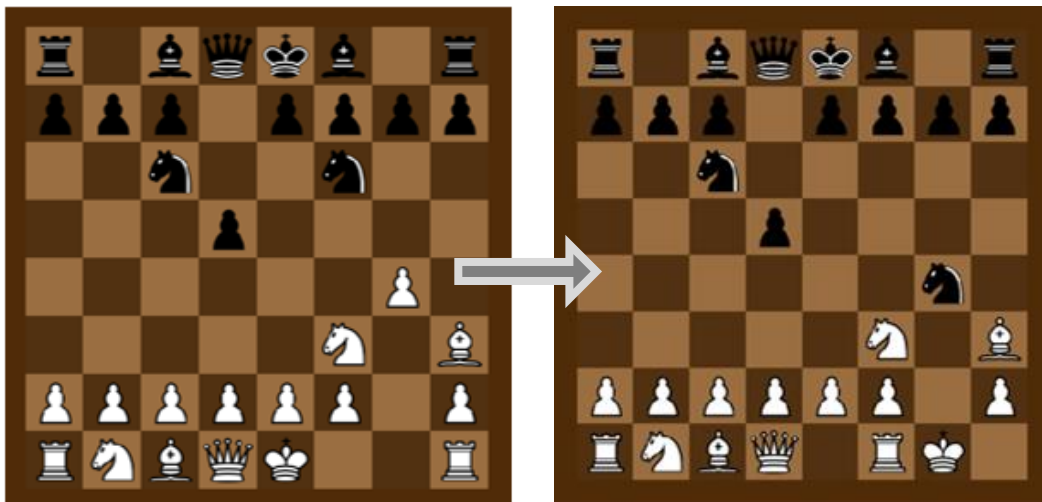


Screenshot 2.0



```
Promote to : ROOK(1), KNIGHT(2), BISHOP(3) or QUEEN(4)
↓
Disposed ChessPiece Texture chess_pieces/wpawn.png
Clearing positions
```

Screenshot 2.1



Screenshot 2.2



Screenshot 2.3



Screenshot 2.4



Screenshot 2.7

AI moved BBISHOP to G4
AI moved BBISHOP to D1
AI moved BPAWN to E6

Screenshot 2.8

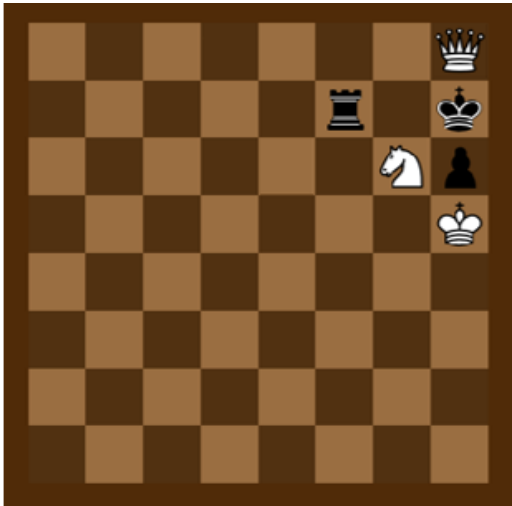
AI moved BKNIGHT to C6

Screenshot 2.9

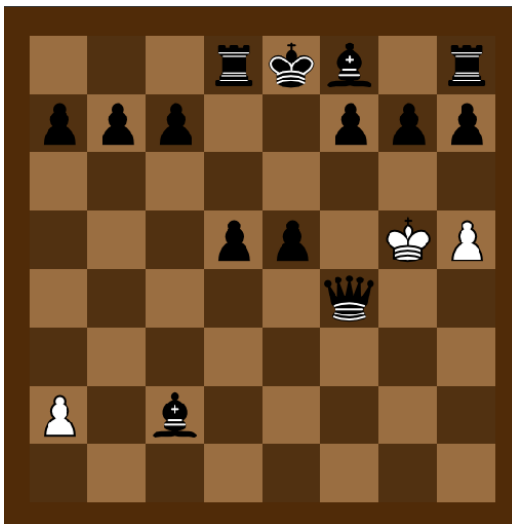


AI moved BKNIGHT to E6
AI moved BKNIGHT to F6
Your king is in check!

Screenshot 3.0



Screenshot 3.1



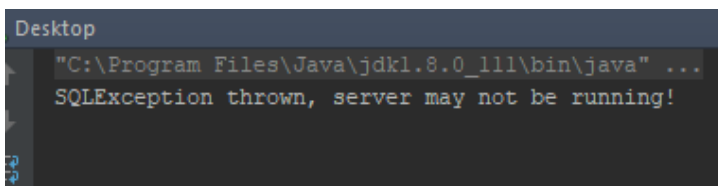
Screenshot 3.2

Game Saved!
Next pawn promotion: Rook

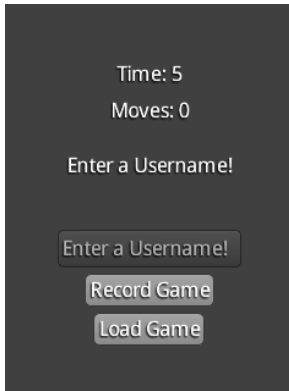
Screenshot 3.3



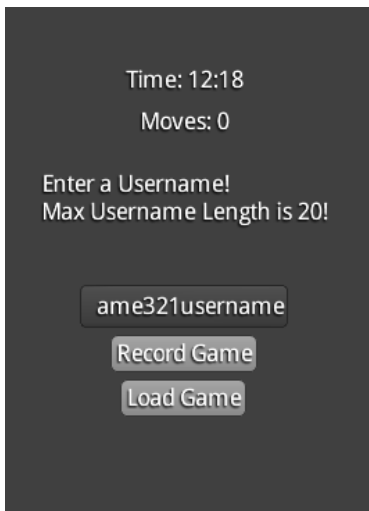
Screenshot 3.4



Screenshot 3.5



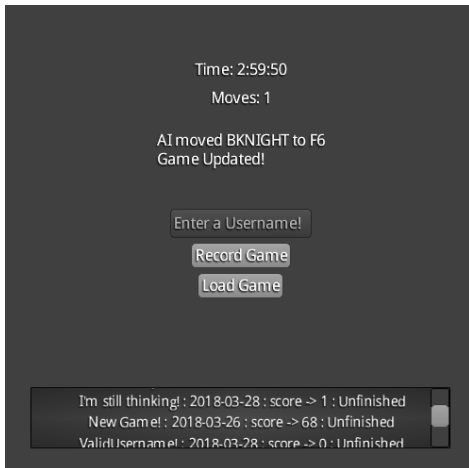
Screenshot 3.6



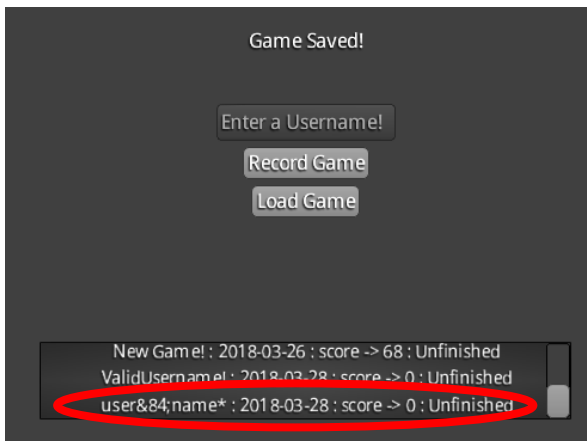
Screenshot 3.7



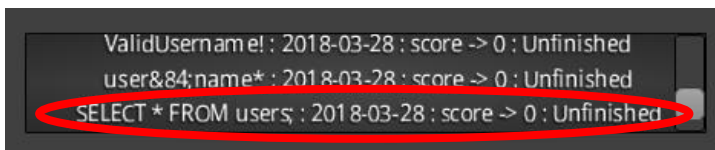
Screenshot 3.8



Screenshot 3.9



Screenshot 4.0



Screenshot 4.1

	id	name	high_score
<input type="checkbox"/> Edit Copy Delete	4	Dylan	0

Screenshot 4.2

	id	time_stamp	time_survived	score	move_history	user_id	win_loss
<input type="checkbox"/> Edit Copy Delete	4	2018-03-25	203	14	61621725313367554142464521235713203177571112133530...	4	0

Screenshot 4.3

	id	time_stamp	time_survived	score	move_history	user_id	win_loss
<input type="checkbox"/> Edit Copy Delete	4	2018-03-25	203	14	61621725313367554142464521235713203177571112133530...	4	0



	id	time_stamp	time_survived	score	move_history	user_id	win_loss
□ Edit Copy Delete	4	2018-03-25	249	20	61621725313367554142464521235713203177571112133530...	4	0

Minimax Stats

2.5

These are the evaluations at the leaf nodes after one move (one method call) at search depths of 2 and 3.

1. Search depth 2, evaluation at leaf node: -670
2. Search depth 2, evaluation at leaf node: -695
3. Search depth 2, evaluation at leaf node: -700
4. Search depth 2, evaluation at leaf node: -695
5. Search depth 2, evaluation at leaf node: -710
6. Search depth 2, evaluation at leaf node: -715
7. Search depth 2, evaluation at leaf node: -810
8. Search depth 2, evaluation at leaf node: -820
9. Search depth 2, evaluation at leaf node: -855
10. Search depth 2, evaluation at leaf node: -850
11. Search depth 2, evaluation at leaf node: -680
12. Search depth 2, evaluation at leaf node: -680
13. Search depth 2, evaluation at leaf node: -700
14. Search depth 2, evaluation at leaf node: -725
15. Search depth 2, evaluation at leaf node: -670
16. Search depth 2, evaluation at leaf node: -665
17. Search depth 2, evaluation at leaf node: -805
18. Search depth 2, evaluation at leaf node: -735
19. Search depth 2, evaluation at leaf node: -735
20. Search depth 2, evaluation at leaf node: -805

1. Search depth 3, evaluation at leaf node: 725
2. Search depth 3, evaluation at leaf node: 655
3. Search depth 3, evaluation at leaf node: 670
4. Search depth 3, evaluation at leaf node: 725
5. Search depth 3, evaluation at leaf node: 640
6. Search depth 3, evaluation at leaf node: 730
7. Search depth 3, evaluation at leaf node: 630
8. Search depth 3, evaluation at leaf node: 655
9. Search depth 3, evaluation at leaf node: 645
10. Search depth 3, evaluation at leaf node: 635
11. Search depth 3, evaluation at leaf node: 640
12. Search depth 3, evaluation at leaf node: 665
13. Search depth 3, evaluation at leaf node: 770
14. Search depth 3, evaluation at leaf node: 805
15. Search depth 3, evaluation at leaf node: 815

16. Search depth 3, evaluation at leaf node: 835
17. Search depth 3, evaluation at leaf node: 610
18. Search depth 3, evaluation at leaf node: 625
19. Search depth 3, evaluation at leaf node: 645
20. Search depth 3, evaluation at leaf node: 655
21. Search depth 3, evaluation at leaf node: 640
22. Search depth 3, evaluation at leaf node: 835
23. Search depth 3, evaluation at leaf node: 750
24. Search depth 3, evaluation at leaf node: 680
25. Search depth 3, evaluation at leaf node: 695
26. Search depth 3, evaluation at leaf node: 750
27. Search depth 3, evaluation at leaf node: 665
28. Search depth 3, evaluation at leaf node: 755
29. Search depth 3, evaluation at leaf node: 655
30. Search depth 3, evaluation at leaf node: 650
31. Search depth 3, evaluation at leaf node: 670
32. Search depth 3, evaluation at leaf node: 675
33. Search depth 3, evaluation at leaf node: 665
34. Search depth 3, evaluation at leaf node: 690
35. Search depth 3, evaluation at leaf node: 795
36. Search depth 3, evaluation at leaf node: 830
37. Search depth 3, evaluation at leaf node: 840
38. Search depth 3, evaluation at leaf node: 860
39. Search depth 3, evaluation at leaf node: 635
40. Search depth 3, evaluation at leaf node: 650
41. Search depth 3, evaluation at leaf node: 670
42. Search depth 3, evaluation at leaf node: 680
43. Search depth 3, evaluation at leaf node: 665
44. Search depth 3, evaluation at leaf node: 860
45. Search depth 3, evaluation at leaf node: 745
46. Search depth 3, evaluation at leaf node: 675
47. Search depth 3, evaluation at leaf node: 690
48. Search depth 3, evaluation at leaf node: 745
49. Search depth 3, evaluation at leaf node: 660
50. Search depth 3, evaluation at leaf node: 750
51. Search depth 3, evaluation at leaf node: 650
52. Search depth 3, evaluation at leaf node: 675
53. Search depth 3, evaluation at leaf node: 665
54. Search depth 3, evaluation at leaf node: 670
55. Search depth 3, evaluation at leaf node: 660
56. Search depth 3, evaluation at leaf node: 685
57. Search depth 3, evaluation at leaf node: 790
58. Search depth 3, evaluation at leaf node: 825
59. Search depth 3, evaluation at leaf node: 835
60. Search depth 3, evaluation at leaf node: 855
61. Search depth 3, evaluation at leaf node: 630
62. Search depth 3, evaluation at leaf node: 655

63. Search depth 3, evaluation at leaf node: 665
64. Search depth 3, evaluation at leaf node: 670
65. Search depth 3, evaluation at leaf node: 670
66. Search depth 3, evaluation at leaf node: 855
67. Search depth 3, evaluation at leaf node: 740
68. Search depth 3, evaluation at leaf node: 670
69. Search depth 3, evaluation at leaf node: 685
70. Search depth 3, evaluation at leaf node: 740
71. Search depth 3, evaluation at leaf node: 655
72. Search depth 3, evaluation at leaf node: 745
73. Search depth 3, evaluation at leaf node: 645
74. Search depth 3, evaluation at leaf node: 635
75. Search depth 3, evaluation at leaf node: 660
76. Search depth 3, evaluation at leaf node: 635

(There were more evaluations for depth of 3, however this is enough to show for testing).

2.6

Each execution value corresponds to how long the method took to produce a best move in milliseconds. This was repeated for 20 moves for both methods...

1. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 209
 2. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 85
 3. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 139
 4. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 216
 5. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 163
 6. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 159
 7. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 167
 8. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 221
 9. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 287
 10. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 140
 11. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 108
 12. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 183
 13. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 126
 14. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 218
 15. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 203
 16. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 117
 17. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 98
 18. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 179
 19. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 90
 20. Execution time for MiniMax with Alpha-Beta pruning algorithm in milliseconds -> 59
-
1. Execution time for MiniMax algorithm in milliseconds -> 2043

2. Execution time for MiniMax algorithm in milliseconds -> 1003
3. Execution time for MiniMax algorithm in milliseconds -> 637
4. Execution time for MiniMax algorithm in milliseconds -> 878
5. Execution time for MiniMax algorithm in milliseconds -> 831
6. Execution time for MiniMax algorithm in milliseconds -> 1047
7. Execution time for MiniMax algorithm in milliseconds -> 1241
8. Execution time for MiniMax algorithm in milliseconds -> 824
9. Execution time for MiniMax algorithm in milliseconds -> 1450
10. Execution time for MiniMax algorithm in milliseconds -> 1728
11. Execution time for MiniMax algorithm in milliseconds -> 1911
12. Execution time for MiniMax algorithm in milliseconds -> 2546
13. Execution time for MiniMax algorithm in milliseconds -> 1704
14. Execution time for MiniMax algorithm in milliseconds -> 1692
15. Execution time for MiniMax algorithm in milliseconds -> 2224
16. Execution time for MiniMax algorithm in milliseconds -> 1865
17. Execution time for MiniMax algorithm in milliseconds -> 2797
18. Execution time for MiniMax algorithm in milliseconds -> 1827
19. Execution time for MiniMax algorithm in milliseconds -> 1619
20. Execution time for MiniMax algorithm in milliseconds -> 2142

The number of positions evaluated for both the minimax and alpha-beta after every method call for 20 moves...

1. Positions evaluated in MiniMax with Alpha-Beta pruning -> 1222
2. Positions evaluated in MiniMax with Alpha-Beta pruning -> 714
3. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2847
4. Positions evaluated in MiniMax with Alpha-Beta pruning -> 1507
5. Positions evaluated in MiniMax with Alpha-Beta pruning -> 1457
6. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2243
7. Positions evaluated in MiniMax with Alpha-Beta pruning -> 1552
8. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2892
9. Positions evaluated in MiniMax with Alpha-Beta pruning -> 3193
10. Positions evaluated in MiniMax with Alpha-Beta pruning -> 3966
11. Positions evaluated in MiniMax with Alpha-Beta pruning -> 5491
12. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2692
13. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2859
14. Positions evaluated in MiniMax with Alpha-Beta pruning -> 1832
15. Positions evaluated in MiniMax with Alpha-Beta pruning -> 2438
16. Positions evaluated in MiniMax with Alpha-Beta pruning -> 4116
17. Positions evaluated in MiniMax with Alpha-Beta pruning -> 464
18. Positions evaluated in MiniMax with Alpha-Beta pruning -> 758
19. Positions evaluated in MiniMax with Alpha-Beta pruning -> 629
20. Positions evaluated in MiniMax with Alpha-Beta pruning -> 709

1. Positions evaluated in MiniMax -> 12435
2. Positions evaluated in MiniMax -> 18561
3. Positions evaluated in MiniMax -> 18273

4. Positions evaluated in MiniMax -> 13492
5. Positions evaluated in MiniMax -> 13688
6. Positions evaluated in MiniMax -> 14332
7. Positions evaluated in MiniMax -> 21828
8. Positions evaluated in MiniMax -> 32631
9. Positions evaluated in MiniMax -> 28215
10. Positions evaluated in MiniMax -> 7320
11. Positions evaluated in MiniMax -> 15334
12. Positions evaluated in MiniMax -> 15683
13. Positions evaluated in MiniMax -> 16634
14. Positions evaluated in MiniMax -> 12479
15. Positions evaluated in MiniMax -> 13810
16. Positions evaluated in MiniMax -> 7717
17. Positions evaluated in MiniMax -> 8039
18. Positions evaluated in MiniMax -> 5934
19. Positions evaluated in MiniMax -> 4734
20. Positions evaluated in MiniMax -> 4009

Evaluation

Overview

At the start of this project, I set out to design and build a chess game with an artificial intelligence that the player could compete against. I have managed to achieve this and have met most of the following criteria I set myself at the start of the project:

- The game board will have to be rendered on screen with all chess pieces in the correct place.
- The movement of each chess piece will have to abide by the official rules of chess.
- The user interacts with the board by using a mouse.
- The algorithm used to calculate the best move will be the MiniMax algorithm using alpha-beta pruning.
- Information stored in the database will include move history and the player high score (number of moves taken to check mate the AI).
- There will be functionality to play back a previously saved game.
- Along with the board, information on the time elapsed since the start of the game and captured pieces will be displayed on screen.
- The game will allow both the player and AI to perform a castling move.
- The game will allow both the player and AI to perform a pawn en-passant move.

I began by starting out with a set of objectives for myself, identifying what the program will do, should do, could do and wouldn't do. Throughout the course of this project, I have managed to meet most of these objectives. Because of that, I would consider this project a success. The following table shows all of these objectives and explains whether I met them and how I met them.

Objective	Met?	How? / Comment / Improvement
When the program is first run, it must render a chess board	Yes.	I placed a set of chess piece objects (ChessPiece object) in

on screen along with all the chess pieces in the correct place.		an 8x8 2D board array. This way I could properly represent a chessboard with occupied and empty positions. I then used a set of chess piece graphics and board tiles to render a chessboard with pieces on screen.
The program must allow the user to move their chess pieces on the board according to the rules of chess.	Yes.	I did this by generating a set of available position for each chess piece that they could move to.
The program must calculate a next move based upon the current state of the board using the MiniMax algorithm and alpha-beta pruning.	Yes.	The game calculates a best move using the minimax algorithm, I also found that there was a noticeable improvement in speed of evaluation when implementing alpha-beta pruning for my program.
There should be functionality for loading previously played games to rewatch, or continue playing if the game isn't finished. This data should be stored in a database and updated after every game.	Mostly.	While I have managed to implement game saves and storing data about a game in a database, I did originally have the intent to play back old games. All the program does currently is display the state of the game at the last save. An improvement therefore could be to build in functionality to play back past games showing each new game state move by move.
There should be a load game button on screen to load any previous game.	Yes.	I needed a way to display the old games played in order for the user to select a game to continue playing.
The rule of castling should be implemented with both the user and AI performing the move.	Yes.	This was one of the more difficult parts of building in the functionality for the game. This was due to the various conditions that must be met in order to castle a rook and king.
The game could implement pawn en-passant. This move doesn't occur very often within a game of chess and so implementing it won't be a necessity.	No.	Unfortunately, I could not implement this functionality within the given time frame. An improvement I could have made therefore would be to implement this in the game.

<p>Tiles could change colour when selected or hovered over.</p>	<p>Yes.</p>	<p>One of the first objectives I met was to allow the player to see which piece is currently selected and which tile is being hovered over. I found it necessary to do this as responses I got from people playing the game mentioned that this would help. An improvement I could make would be to highlight all available moves for a piece currently being selected by the player.</p>
<p>I could implement a main menu system (This would be in the form of a separate screen).</p>	<p>No.</p>	<p>Though all of the functionality of that would have been in the main menu system was implemented (save game, load game, etc), I did not manage to implement a main menu system. Another improvement I could make would therefore be to add in a main menu system with options such as new game, load game, view scoreboard, etc.</p>
<p>The game could display data on the side of the screen about time elapsed for the player, how many moves have been made, taken pieces and a message box to prompt the user on the move made by the AI, whether the king is in check or there is a checkmate, which piece a pawn was just promoted to, etc.</p>	<p>Yes.</p>	<p>I found that storing information such as pieces lost was essential. I did this by using a table widget to lay out the HUD.</p>

Feedback

I have received feedback about the game from both the client as well as users of the software. There were many different proposed improvements as well as what was liked about the software. Here I intend to speak about all feedback that I got.

Ease of use

Most of the feedback I received about the playability and ease of use of the software was positive. Users mentioned that I was difficult to click a button or move a piece in an incorrect way and crash the program. The feedback I received highlighted that the controls and display was easy to understand. An example of an improvement mentioned to me by a user was that tiles could be highlighted for a selected piece's available moves.

Criticism

There were several users who mentioned that they did not like how cluttered the screen was. Though they understood what each widget in the HUD did, they did not like the layout. They mentioned it would be a good idea to build a main menu system or perhaps have a drop-down menu at the top of the screen to only show information when the user selects it.

Extensions

The client spoke about extensions to the software that could be made in the future. They said that the game could include a multiplayer system where people could play against one another both locally on the same machine as well as across a network.

Improvements

Because of the time constraints for my project, I was limited to how much I could implement. There are a number of improvements or extensions I could build into the system given a few more months.

Improvement	Description
Build in a multiplayer system for 2 player functionalities.	Since my program already has the functionality built in for checking a move is valid, this would not be too difficult to implement, nor would it take very long. This would improve the current system as it would allow the player to compete against an opponent that responds in a different way to an AI.
Build in a multiplayer system for competing with other players over a network.	This would be the most difficult to implement and would require lots of time. It would go about doing this by including a client-server model which would connect different players together. This would improve the system by allowing the player to play a game with someone who may be in a separate location on a separate machine.
Add in functionality for allowing the AI to learn from past games.	This would require me to use the information about each game that I have stored in the database. I would use this information as a factor for determining what the best move to make would be. This would improve the software as it would allow the AI to slowly become more experienced and use strategy in a game rather than determining the best move based on a heuristic evaluation of the board.
Add in a main menu system including new features such as viewing a users high score table.	At the start of my project, I wanted to implement a stored procedure to update the users table with their latest high score. However, I did not get around to implementing this as I had run out of time. An improvement therefore would be to create a high score table of users in a main menu system. This would not only help users keep track of their high score but also where there ranking would be amongst other users. The main menu system would

	improve the system as it would allow the user to more easily navigate the game.
Add in the ability to change the AI difficulty level.	Having an easy, medium and hard difficulty would improve the current system as it would allow users to adjust the game difficulty to match theirs. This way, a game won't be too hard or too easy for any particular user.