Product Specification

Group 6

May 10, 2018

Application Objective

The objective of this application is to make the extraordinary world of chess more accessible to the people. No longer do you have to bring a big robust wooden board. You just have to pick up your computer and start playing. Not only can you play by yourself but you can also play against our advanced AI on the level that you prefer. With the magical and outstanding graphics of our app, the game of chess will be more colourful and packed with mind blowing effects.

System Requirement

- Supported Platforms
  - Windows 10 or above
  - Mac OSX High Sierra or above
  - Ubuntu Linux
- Hardware requirements:
  - Processor: 32-bits or greater
  - At least 512 MB RAM
- Software Requirements:
  - JAVA SE 8 or above

Functional requirements

- A human player must be able to play against an AI player (simple, intermediate)
- Support standard chess rules
- Keep track of ranking (winner statistics)
- Enable users to create an account (to keep track of games won/lost)

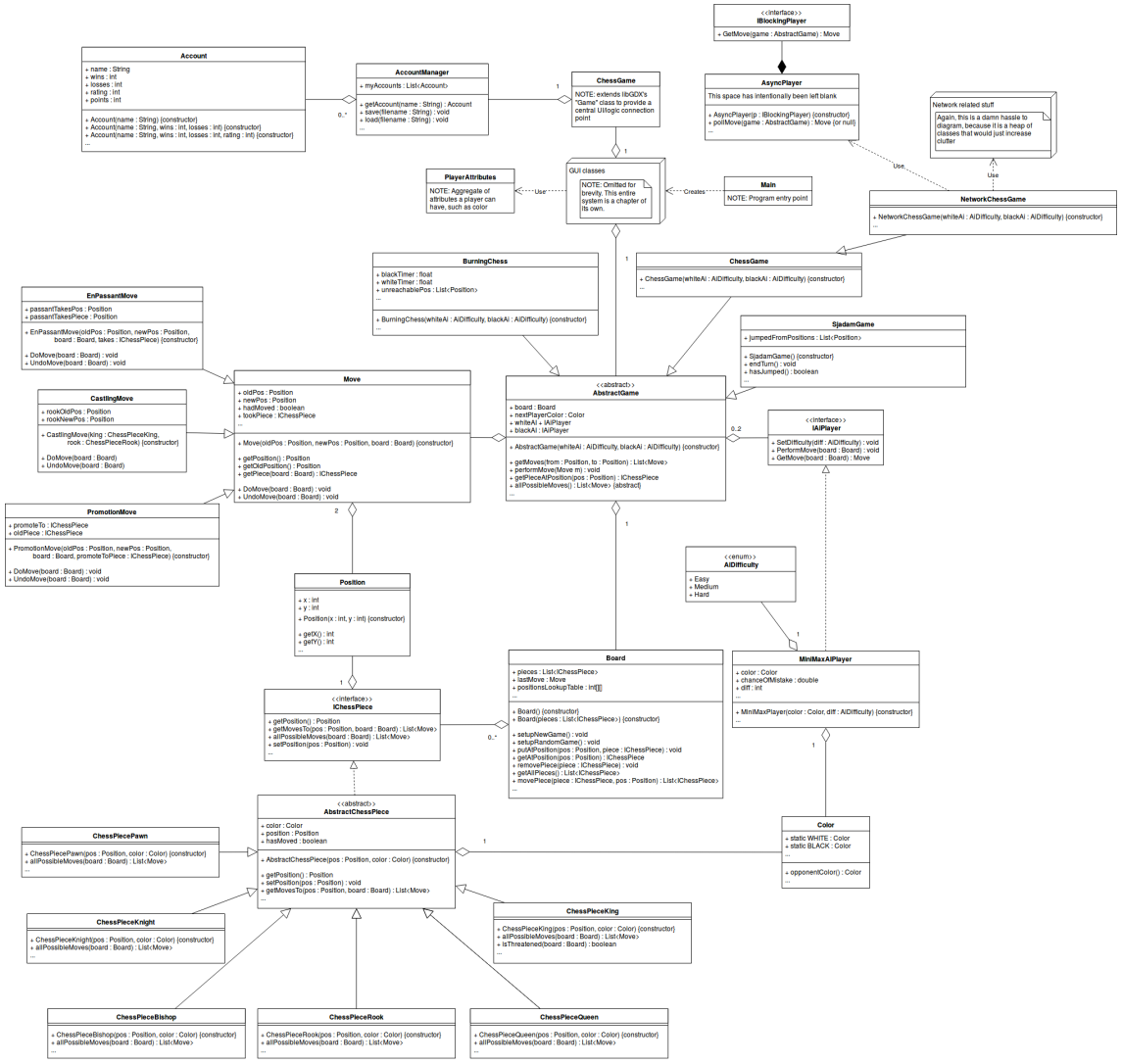
Non-functional requirements

- The GUI must be easy to use and visually appealing
- All images and other graphics used must have an open data license
- All source code and build script must have an open source license
- The Java code shall be documented according to best practices using JavaDoc
- The simple machine player must make a move within 1 second
- The intermediate machine player must make a move within 3 seconds
- The application must be easy to extend, to e.g. support different rule sets for Chess

User Stories

- As a chess player, I expect the chess pieces to behave like they do in regular chess, so that I can get as true a chess playing experience as possible
- As a user, I expect the game to easily be understandable, because that will reduce the required effort to start playing
- As a user, I expect the game to look visually pleasing, because this will make playing it more enjoyable
- As a user, I expect the game to have simple controls, so I can focus on what I want to do and not how to do it
- As a user, I expect to be able to play this on my computer, so I can enjoy a game of chess in my spare time
- As a user, I expect the application to be responsive, so that I can spend more time playing and less time waiting
- As a user, I expect the application to not crash, because software crashes are a source of frustration I'd rather avoid
- As a user, I expect to be able to play against a human player, because I want to enjoy playing chess with my friends
- As a user, I expect to be able to play against a machine player, because I want to be able to play chess even without other humans around

Class Diagram



Explanation

In this project, we've been very insistent on decoupling logic from UI, and as such, the underlying logic is completely independent of the particular way of displaying them. At the very top logical level is the `AbstractGame` class, which contains the board and manages the players. As noted by the name, this class is abstract, and subclasses will override its methods as required to get the specific game version behavior - the standard chess rules are now defined by the class `ChessGame`. The UI uses this class's public API whenever it needs to get some information about the state of the game, be it whose turn it is next, the current state of the board, the piece occupying a given

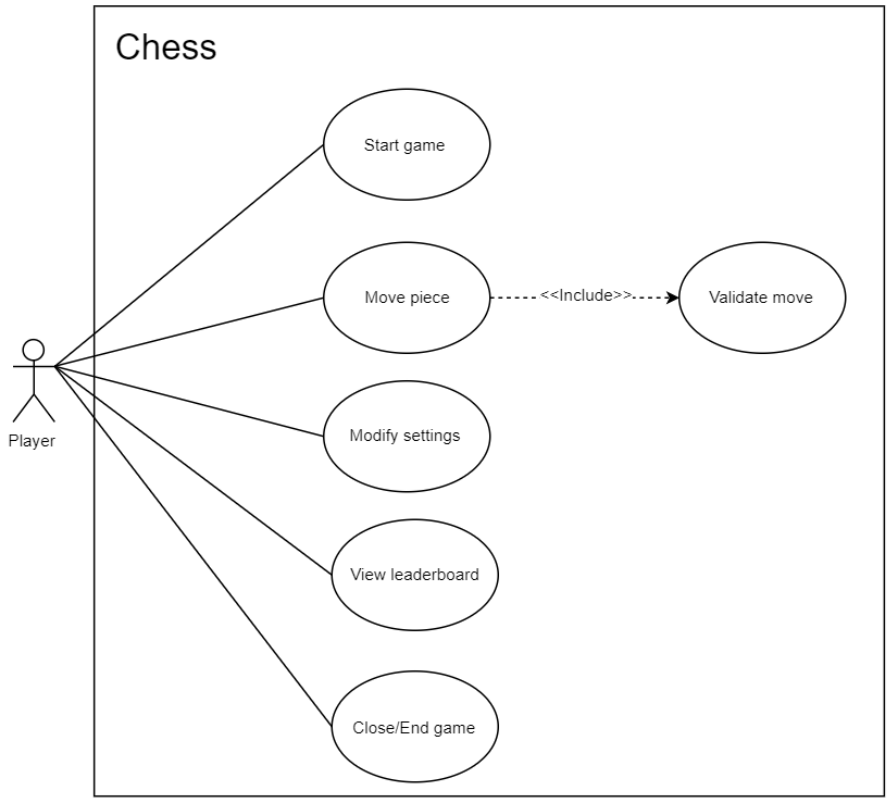
square, etc. The AbstractGame contains the information about the AI players. The algorithm used to choose the next move is contained within the `MiniMaxAIPlayer` class, implementing `IAIPlayer`, and uses minimaxing to a certain lookahead level, coupled with alpha-beta pruning, in order to select the next move to make.

The AbstractGame also contains a `Board`, which consists of a list of `IChessPieces`, which in turn is implemented by concrete kinds of chess pieces. Common functionality for all chess pieces is pushed up into the abstract `AbstractChessPiece`, in order to reduce code duplication. Each piece keeps information about their own position, to reduce the amount of arguments we have to pass around, but it should never be changed by the chess pieces themselves, only by using the public Board API, in order to keep the information consistent. Each chess piece is responsible for determining its own rules regarding which moves they are allowed to make, given the current state of the Board.

Finally, we have a couple of utility classes. The `Move` class is responsible for actually changing the Board state - this happens in the `DoMove` and `UndoMove` methods, which takes a Board and does the required changes to either perform a move, or to revert it. This is the bread and butter of the AI player, which needs to be able to simulate a lot of moves in order to determine what the best course of action is, and subsequently return the Board to its original state. The special moves in the game, `CastlingMove` and `EnPassantMove`, override these two methods in order to avoid corrupting the board state. The fact that a lot of the logic is pushed out to the edges (the pieces and moves) makes it easy to extend the game by instructing it to run on different pieces and different moves obeying a different ruleset. Finally, the positions are encapsulated in the `Position` class which, to spare our team a world of headaches, is immutable.

TL;DR: The UI queries the AbstractGame for game state info, which contains a Board it instructs to move pieces to given positions. The pieces are responsible for determining where they can go, and the Move class makes the changes to the Board state take effect. The game also optionally manages AI players which use minimaxing to find the best move when looking a certain number of steps ahead.

User Case Diagram



Fully dressed use case #1: Move a piece

Use case name: Move a piece

Scope: Chess application gameplay

Level: User goal

Primary actor: Current player

Stakeholders and interests:

- **Current player:** Wants to move a piece to a different square
- **Chess application:** Must restrict movement to a valid square

- **Opponent player:** Wants to be in a good position for their next move

Preconditions: It's current player's turn to move. Previous moves have been logged. Previous moves have been valid.

Success guarantee (postconditions): Exactly one of current player's pieces is in a different position. The move has been recorded. The recorded move was valid. It's opponent player's turn to move, or the game is over.

Main success scenario:

1. Current player selects piece to move
2. Chess application indicates possible squares the player can move to
3. Current player selects one of the highlighted squares
4. Chess application validates move
5. Chess application performs move
6. Chess application checks for promotion
7. Chess application logs move
8. Chess application checks if the move resulted in check
9. Chess application displays the result
10. Chess application changes turn from current player to opponent player

Extensions:

*a. At any time, current player may exit the game
    1. Chess application aborts current game
    2. On next startup, chess application displays an option to continue previous game
        *a.2a. The player selects to continue the game
            1. The game is resumed from the start of the aborted turn
        *a.2b. The player starts a new game
            1. The aborted game is discarded

*b. At any time during their turn, the player may resign the game
    1. Chess application asks for confirmation
        *b.1a. Current player confirms resignation
            1. Chess application records resignation
            2. Chess application ends current game with opponent player as the victor
            3. Chess application performs player ranking calculations
            4. Chess application displays the information that the player has resigned

5. Current player exits the game

*b.1b. Current player cancels resignation

1. Resume turn from last step

1a. Current player selects one of opponent player's pieces, or a square with no piece on it

1. Chess application ignores the input

1b. Current player selects a piece which cannot move under the current conditions

1. Chess application displays a notice saying the piece cannot move

1b.1a. Current player deselects piece

1. Resume turn from step 1

1b.1b. Current player selects a different piece which cannot move

1. Repeat step 1b

1b.1c. Current player selects a different piece which can move

1. Resume turn from step 2

3a. Current player selects a square which is not indicated

1. Chess application ignores the input

3b. Current player selects a different piece

1. Resume turn from step 1 or any step 1 extension as appropriate

4a. The move is invalid (should not happen under normal circumstances)

1. Chess application displays an error message
2. Chess application sends anonymous error report to developers
3. Resume turn from step 1

5a. The move was to an empty square

1. Chess application updates position of moved piece on board
2. Resume turn from step 6

5b. The move was to a square containing an opponent player's piece

1. Chess application removes opponent player's piece from board
2. Chess application updates position of moved piece on board
3. Resume turn from step 6

6a. The move resulted in a promotion

1. Chess application replaces pawn with queen

8a. The move results in check

1. Chess application shows notice that opponent player is in check

8b. The move results in check mate

1. Chess application shows notice that opponent player is in check mate
2. Chess application ends current game with current player as the victor

3. Chess application performs player ranking calculations

10a. The game has ended

1. Chess application does not start opponent player's turn
2. Chess application displays option to exit the game

Special requirements:

- Mouse or keyboard input devices for player input
- Display screen for output
- Host computer needs a graphical user interface

Technology and data variations list:

- Piece selection using mouse or keyboard input
- New position selection using mouse or keyboard input
- Store moves as Move object for short term storage due to ease of use in-application, or store moves as String for long term storage between sessions due to storage space gains

Frequency of occurrence: Ranges from every few seconds to several minutes

Open issues:

- Special indication of special moves (castling, en passant) to players?
- Step 6: Part of "perform move" or a separate step?
- Special indication of the king being in check?
- Special indication of move inavailability **due to** the king being in check?
- Step 1b: If no available moves indicated, there may not be a need for a notice explaining this
- Step 5a en passant special case: Pawn moves to empty square but still captures piece
- Step 8 (or maybe step "0"): Other potential game-ending situations, such as:
  - Player is in check but cannot move (stalemate)
  - Insufficient material (stalemate)

Fully dressed use case #2: Compute available moves

Use case name: Compute available moves

Scope: Chess application gameplay

Level: Subfunction

Primary actor: Chess application

Stakeholders and interests:

- **Current player:** Needs to know available moves
- **Chess application:** Must restrict movement to a valid square

Preconditions: Information about position of pieces is available. Information about previous moves is available. Previous moves have been valid. Previous moves are in chronological order. Information about which piece or which player to compute the moves for is available.

Success guarantee (postconditions): Preconditions still hold. A complete list of available moves has been computed. No move in the list is invalid given the current state of the board and move history.

Main success scenario:

1. Check whether king of current player is in check
2. Check whether castling is possible, if applicable
3. Check whether en passant is possible, if applicable
4. Compute list of all possible places piece could go on an empty board
5. Eliminate positions occupied by current player pieces
6. Eliminate positions blocked by current player or opponent player pieces
7. Return computed moves

Extensions:

1a. King is in check
    1. Compute moves for relevant pieces
        1a.1a. Information about current player is available
            1. Perform steps 1a.1b.1-4
            2. Perform steps 1a.1c.1-5 for all other current player pieces
        1a.1b. Information about piece to move is available, king is selected
            1. Compute places king could go on an empty board
            2. Eliminate positions occupied by current player pieces
            3. Eliminate positions threatened by opponent player pieces
            4. Compute pieces which can block the piece threatening the king
        1a.1c. Information about piece to move is available, other piece selected

1. Compute positions piece could go on an empty board
2. Eliminate positions occupied by current player pieces
3. Eliminate positions blocked by current player or opponent player pieces
4. Eliminate positions which open up a different line of attack on the king
5. Eliminate positions which do not block the current line of attack on the king

2. Resume from step 7

2a. King has previously moved

1. Resume froms step 3

2b. King has not previously moved

1. Check whether castles have moved

2b.1a. Both castles have moved

1. Resume froms step 3

2b.1b. At least one castle has not moved

1. Check for all unmoved castles if path is clear
2. If path blocked by other pieces, do not add castling move for this castle
3. If part of path threatened by opponent player piece, do not add castling move for this castle
4. If path is clear, add castling move to list

2a. Information about piece to move is available, piece is not king

1. Do not check if castling is possible

3a. Information about current player is available

1. Perform step 3b.1 for all pawns of current player

3b. Information about piece to move is available, piece is pawn

1. Check if en passant can be performed

3b.1a. Pawn is not on 5th rank

1. Do not add en passant move for this pawn

3b.1b. No piece on adjacent files is opponent player pawn

1. Do not add en passant move for this pawn

3b.1c. Last move in move history was not double step forward by adjacent pawn

1. Do not add en passant move for this pawn

3b.1d. En passant can be performed

1. Add en passant move for this pawn to list

3c. Information about piece to move is available, piece is not pawn

1. Do not check if en passant is possible

Special requirements: Computation should be as fast as possible, in order to create the best possible user experience.

Technology and data variations list:

- Version which computes all possible moves of a given player
- Version which computes all possible moves for a given piece

Frequency of occurrence: Computation for all possible moves of a player will normally only happen in order to compute possible moves for AI, the frequency of which ranges from every few seconds to several minutes depending on how long the human player takes to make a move. Computation for a specific piece may be very frequent during player turns, perhaps continual.

Open issues:

- Step 1a.1b.3: When king moves, position which was not previously threatened due to being blocked by king may become threatened
- Computationally intensive operations such as modifying list of moves or looping through move history may need shortcuts to decrease computation time, including
- Retrieve moves and eliminate ineligible moves as a single step
  - Shortcuts when checking ineligibility: for instance, if one horizontal line of a bishop
  - is blocked, do not evaluate eligibility of all subsequent positions on that line
- Shortcuts for determining whether rooks or king have been moved may need to be devised