Software Engineering GP02 Project

Design Specification

|  |  |
| --- | --- |
| Authors: | Jack Book [jab153], Thomas Roethenbaugh [tpr3], Will Abbott [wia14], Michael Stamp [mjs36], Micah Barendse [mib60] |
| Config Ref: | SE.G02.DesignSpec |
| Date: | 27th April 2023 |
| Version: | 1.1.1 |
| Status: | For Review |

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Copyright © Aberystwyth University 2023

**CONTENTS**

1. Introduction 3

1.1 Purpose of this Document 3

1.2 Scope 3

1.3 Objectives 3

2. Decomposition Description 3

2.1 Programs in the System 3

2.2 Significant Classes 3

2.3 Requirements to Classes Mapping 4

3. Dependency Description 5

4. Interface Description 5

4.1 Main 5

4.2 Game Package 5

4.3 Pieces Package 8

4.4 Moves Package 10

4.5 GUI Package 13

4.6 Game Save Package 13

4.7 Util Package 13

5. Detailed Design 14

5.1 UML Sequence Diagrams 14

5.2 UML Class Diagram 15

5.3 Significant Algorithms 16

5.4 Significant data structures 18

REFERENCES 18

DOCUMENT HISTORY 19

# Introduction

## Purpose of this Document

The purpose of this document is to give insight into how the game will be designed. It will describe the decomposition of the programs and the modules that fall within them, including any significant classes and how the classes overlap with the functional requirements. It will also describe the relationships and dependencies the modules have between each other. Additionally, it will provide a description of the interfaces used and how programmers or testers will use the facilities provided by a module. Finally, it will lay out the internal details of any modules that are not obvious to an external reader [1].

## Scope

The document should be read by all members of the project. Readers should have a good understanding of the User Interface Specification document [3].

## Objectives

The objective of this document is to allow the reader to understand the inner mechanisms of the classes used in the project and go into further detail of how more complicated classes work.

# Decomposition Description

## Programs in the System

The system as designed only contains one program and is run as a JVM process. This singular program handles both the logic and graphics of the game in several packages and classes. The backend logic relies on multiple packages, notably the game, moves, and pieces packages. These packages contain Java classes which are used as objects by the final program. The frontend relies on the GUI package, the classes utilizing JavaFX. The saving and loading functionalities are isolated in the gameSave package. Further, all JUnit classes are contained in the tests package.

## Significant Classes

The program is broken down into separate packages, each package containing relevant classes and materials needed to run certain aspects of the program. Each of the sections below will describe the packages, and briefly explain the classes used.

### game

The game package contains four classes – Game, Board, Square, and Player, these are used to build the game and holds the majority of the backend structure.

**Game -** collects the relevant information from other classes in the package and holds the properties of the current game in progress.

**Board** – stores the chessboard, which is an array of Squares, and contains methods directly related to the overall chessboard, such as creating the board at the start of the game.

**Square** - is responsible for looking after each cell on the chessboard and what those cells contain.

**Player** - collects the information regarding each player, e.g., name, color, and captured pieces.

### pieces

The pieces package has seven classes – Piece, Bishop, King, Knight, Pawn, Queen, and Rook. These classes contain the material for each specific piece. The individual classes representing each piece work in similar ways, apart from the valid moves method that determines what each valid move for the pieces are and allows the movement to take place. Piece is an abstract class that is inherited by the others.

### moves

The moves package contains five classes – MakeMove, CheckChecker, Castle, EnPassant, and Promote. Castle, EnPassant, and Promote classes are considered ‘special moves’ that are not called as often as the other classes; as such, they are in an internal package.

**MakeMove –** This class holds the methods for moving a piece on the chessboard. It contains methods that are specifically used for making the moves that involve EnPassant and Castling.

**CheckChecker** – This class is responsible for checking whether either of the King’s is currently in check or checkmate.

**Castle** – This class is for checking if either the Black King or White King can Castle.

**EnPassant** – This class holds the method to get the valid moves for enPassant.

**Promote** – This class promotes a Pawn to the specified piece type.

### GUI

This package includes all the frontend classes, as well as all images required by the program, such as the menu background and all the individual pieces.

**GameGUI –** This is the class that runs the main graphical user interface for the game. This class will show the game in progress and display the game in the correct manner, for the game to run correctly, in accordance with the users’ inputs. It includes 2 methods. The first initializes the main menu, displaying the text boxes for the users’ names, the selection box for assigning the user colors, and any buttons that are needed in accordance to the User Interface Design Specification.[2]

**BoardGUI** -

### util

The classes within this package provide simple utilizes to other package classes. For this program, these classes define the colors and pieces available within the chess game.

**Color –** This class simply defines the two available colors within chess: Black and White.

**Type –** This class defines the pieces available within chess. These pieces are King, Queen, Rook, Bishop, Knight and Pawn.

### gamesave

There is one class in this package – GameSaveManager. This class contains the functions to save and load ongoing games and replays.

**GameSaveManager** – Loads a specific round of a saved replay.

**FilePicker** – A class for opening a directory picker and returning a file.

## Requirements to Classes Mapping

|  |  |
| --- | --- |
| Functional Requirements | Classes providing requirement |
| FR1 | Board, Game, Player, Square |
| FR2 | Player, Game |
| FR3 | Board |
| FR4 | makeMove, Bishop, King, Knight, Pawn, Piece, Queen, Rook |
| FR5 | makeMove, Bishop, King, Knight, Pawn, Piece, Queen, Rook, Castle, enPassant, CheckChecker |
| FR6 | CheckChecker |
| FR7 | CheckChecker |
| FR8 | GameGui, BoardGUI |
| FR9 | GameGui, BoardGUI |
| FR10 | GameGui, GameSaveManager, BoardGUI |
| FR11 | GameGui, GameSaveManager, BoardGUI |

# Dependency Description

Graphical user interface

Description automatically generated with low confidence

Figure 1 - Component Diagram

# Interface Description

## main Package

* Type: Public
* Extends: Application
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| Void | start() | primaryStage; Stage |  |
| Void | main() | args; String[] |  |

## game Package

### Game

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Game() |  | Constructor. Creates a board object, sets the turn to White, and sets the roundCount to 1. |
| Board | getBoard() |  | Returns the board field. |
| void | setWhitePlayer() | name; String | Sets the whitePlayer field to a new player object with the passed String as the name. |
| void | setBlackPlayer() | name; String | Sets the blackPlayer field to a new player object with the passed String as the name. |
| Player | getWhitePlayer() |  | Returns the whitePlayer field. |
| Player | getBlackPlayer() |  | Returns the blackPlayer field. |
| Color | getTurn() |  | Returns the turn field. |
| void | setTurn() | turn; Color | Sets the turn field to the passed Color. |
| int | getRoundCount() |  | Returns the roundCount field. |
| void | nextRound() |  | Increments the roundCount field. |
| String | getSaveDirectory() |  | Returns the saveDirectory field. |
| String | setSaveDirectory() | name; String | Sets the game save directory |

### Board

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Board() |  | Constructor. Creates a new 2D Square object at 8 x 8. Sets the enPassantPiece field to {-1,-1} and then calls the boardReset method. |
| Square[][] | getBoardArray() |  | Returns the boardArray field. |
| int[] | getEnPassantPiece() |  | Returns the enPassantPiece field. |
| void | setEnPassantPiece() | enPassantPiece; int[] | Sets the enPassantPiece field as the passed int array. |
| void | boardReset() |  | Resets the boardArray field to the starting configuration of pieces. |
| void | clearBoard() |  | Clears the boardArray field so new pieces are on the board. |
| void | printBoard() |  | Used for debugging, prints the boardArray field to the console. |
| void | setSquare() | square; Square  row; int  col; int |  |
| Board | clone() |  |  |

### Square

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Square() |  | Constructor. Used for Square’s that do not contain a Piece. Sets hasPiece field to false. |
|  | Square() | piece; Piece | Constructor. Used for Square’s that do contain a Piece. Sets hasPiece field to true. |
| Piece | getPiece() |  | Returns the piece field. |
| boolean | hasPiece() |  | Returns the hasPiece field. |

### Player

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Player() |  |  |
|  | Player() | color; Color,  name; String | Constructor. Sets color and name field as passed values. Initializes takenPieces field as an ArrayList. |
| Color | getColor() |  | Returns the color field. |
| String | getName() |  | Returns the name field. |
| List<Piece> | getTakenPieces() |  | Returns the takenPieces field |
| void | addTakenPieces() | piece; Piece | Adds passed Piece to takenPieces field. |

## pieces Package

### Piece

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Piece() |  |  |
|  | Piece() | color; Color | Constructor. Sets passed color as color field and sets hasMoved field to false. |
| Color | getPieceColor() |  | Returns the color field. |
| Color | getEnemyPieceColor() |  | Using color field, returns opposing sides color. |
| Type | getPieceType() |  | Returns the ID field. |
| boolean | hasMoved() |  | Returns the hasMoved field. |
| void | setHasMoved() |  | Sets the hasMoved field to true. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Abstract method to be redefined by inherited classes. Returns a List<int[]> of valid move coordinates. |
| boolean | addValidMove() | boardArray; Square[][],  piece; Piece,  row; int,  col; int | A method to check if a specific coordinate is valid, if so, it adds it to the returnArray field. |
|  | getPossibleMoves() |  |  |

### Bishop

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Bishop() | color; Color | Constructor. Uses the Piece constructor and sets ID field to Bishop type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the Bishop piece. |

### King

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | King() | color; Color | Constructor. Uses the Piece constructor and sets ID field to King type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the King piece. |

### Knight

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Knight() | color; Color | Constructor. Uses the Piece constructor and sets ID field to Knight type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the Knight piece. |

### Pawn

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Pawn() | color; Color | Constructor. Uses the Piece constructor and sets ID field to Pawn type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the Pawn piece. |

### Queen

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Queen() | color; Color | Constructor. Uses the Piece constructor and sets ID field to Queen type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the Queen piece. |

### Rook

* Type: Public
* Extends: Piece
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | Rook() | color; Color | Constructor. Uses the Piece constructor and sets ID field to Rook type. |
| List<int[]> | getValidMoves() | board; Board,  piece; Piece,  row; int,  col; int | Overrides inherited method with commands to retrieve correct coordinates for the Rook piece. |

## moves Package

### MakeMove

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| void | movePiece() | board; Board,  player; Player,  pieceRow; int,  pieceCol; int,  targetRow; int,  targetCol; int | A method that moves a piece on the board. This method is only for standard move. |
| void | movePiece() | board; Board,  player; Player,  pieceRow; int,  pieceCol; int,  targetRow; int[],  targetCol; int[] | A method designed to move a Pawn if a enPassant move is selected. |
| void | movePiece() | board; Board,  player; Player,  pieceRow; int[],  pieceCol; int[],  targetRow; int[],  targetCol; int[] | A method designed to move a King and Rook if a Casting move is selected. |
| void | clearEnPassantPiece() | board; Board | A method to call the Board class method setEnPassantPiece passing in {-1,-1}, thus resetting the field. |
| void | addTakenPiece() | player; Player,  piece; Piece | A method to add the piece to the players takenPieces list, using the Player class method addTakenPieces. |

### CheckChecker

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| List<int[]> | checkChecker() | board; Board,  row; int,  col; int | A method that returns a list of coordinates of pieces that are attacking the square the King is located on. Returns empty if the King is not in check. |
| boolean | checkMateChecker() | board; Board,  row; int,  col; int | A method that returns a True or False depending on whether the King is in Check Mate. |

### Castle

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| List<int[]> | canCastle() | board; Board,  piece; Piece,  row; int,  col; int | A method to check if a King can castle with a Rook. If so, it returns a List<int[]> containing the valid coordinates. If not, it returns the List empty. |
| int[] | checkKingSide() | board; Board,  piece; Piece,  row; int,  col; int  start; int,  end; int,  direction; int | A method that checks for a Rook on the specified side of the King. This method is exclusively called by canCastle. |

### EnPassant

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| List<List<int[]>> | getMoves() | board; Board,  piece; Piece,  row; int,  col; int | A method to return coordinates to be used when making an EnPassant move. It returns two Lists contained in a single list. The first List contains the coordinates the Pawn will move to. The second List contains the coordinates of the Pawn to be taken. |

### Promote

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| void | promotePawn() | board; Board,  row; int,  col; int,  newPiece; Type | A method to promote a Pawn to a specified piece type. |

## GUI Package

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | GameGui() | stage; Stage | Constructor. |
| Void | initMainMenu(Stage primaryStage) | primaryStage; Stage |  |

### GameGui

* Type: Public
* Extends: Nothing
* Public Methods

### BoardGUI

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
|  | BoardGUI |  | Constructor. |
| Grid Pane | capturedWhitePieces() | newGame; Game | Method for storing white's captured pieces |
| Grid Pane | capturedBlackPieces() | newGame; Game | Method for storing black's captured pieces |
| Grid Pane | boardFunc() | newGame; Game | Method to create the chess board |
| Void | setSavedPieces() | newGame; Game  board; Gridpane | Loads the pieces onto the board from a saved game |
| Void | setStartingPieces() | newGame; Game,  board; GridPane |  |
| Image View | createPiece() | type; String,  color; String | Method to create a new piece for the board |
| Void | setUpPieceHandlers() | newGame; Game,  piece; StackPane,  board; GridPane,  pieces; String,  color; String | Method to add event handlers to the chess board |
| HBox | whoseTurnNameBox() | newGame; Game |  |
| String | whoseTurnName() | newGame; Game | Function to return the name players whose turn it is |
| Boolean | whoseTurn() | newGame; Game,  color; String |  |
| Void | movePiece() | newGame; Game,  selectedPiece; StackPane,  board; GridPane,  destRow; int,  destCol; int,  piece; String |  |
| Void | handleClick() | newGame; Game,  row; int,  col; int,  board; GridPane,  piece; String |  |
| Boolean | capture() |  |  |

* Type: Public
* Extends: Nothing
* Public Methods

## gamesave Package

### GameSaveManager

* Type: Public
* Extends: Nothing
* Public Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Parameters** | **Description** |
| Game | loadReplayRound() | directory; String, round; int | Loads a specific round of a saved replay. |
| Game | loadOngoingGame() | directory; String | Loads an ongoing game. |
| void | saveOngoingGame() | game; Game | Saves an ongoing game. This should be called at the end of every turn. Requires Game.saveDirectory to be set. |
| void | saveReplay() | game; Game | Saves a replay. |

## util Package

### Color

* Type: Public
* Extends: Nothing
* Constants

|  |
| --- |
| **Name** |
| WHITE |
| BLACK |

### Type

* Type: Public
* Extends: Nothing
* Constants

|  |
| --- |
| **Name** |
| KING |
| QUEEN |
| ROOK |
| BISHOP |
| KNIGHT |
| PAWN |

# Detailed Design

## UML Sequence Diagrams

### Main Menu UML (UC01 to UC05)

Diagram

Description automatically generated

Figure 2. Main Menu UML

### Game UML (UC06 to UC19)

Diagram

Description automatically generated

Figure 3 - Game UML

### Move Pieces (UC07, 12, 13)

Box and whisker chart

Description automatically generated with medium confidence

Figure 4 - Move Pieces UML

## UML Class Diagram

This diagram shows how the classes within the Chess Tutor game link between each other, to make the game run correctly.

Diagram

Description automatically generated

Figure 5 - UML Class Diagram

## Significant Algorithms

### Move a Piece

To move a piece on the chessboard the user would first click a piece. It would then display all the valid moves on the screen. This is done by calling the getValidMoves method. This returns coordinates which the GUI would then represent to the user. If the piece is a King or Pawn, it would also call the Castling method and enPassant method respectively. It would then proceed to display these on the GUI. The user can then decide which move they wish to take. Or if they do not want to take any of those moves, click on another piece and repeat the process.

Once they have decided to move a piece, depending on if it is a special move (Castling/enPassant), it will call the correct method. It moves a piece by setting the target square to the value of the original square. It then resets the original square to represent an empty square. If the target square was an enemy piece, it is added to the players taken piece list.

### Castling

The Castling method is separated from the getValidMoves method. Therefore, it has to be specifically called by the program when a King piece is selected.

To Castle a King a user would select their King on the chessboard. It would then run the standard getValidMoves method and the canCastle method. The canCastle method checks first that the King has not moved in the game yet, if so, it checks for a Rook on either side of the King. If there is a piece blocking the King and the Rook, or there is no Rook found, the method returns an empty list. If an unblocked Rook is discovered, the method returns a List with several coordinates; the position the King will move to, the position of the Rook, the position the Rook will move to. If the user accepts the Castling move, then the makeMove method is called which has overloaded methods designed for standard moves as well as Castling moves. This method works in a similar way to the standard makeMove method, clearing the previously occupied squares.

### enPassant

This move requires some unique rules that make it difficult to keep its move set in the Pawn class. The method of determining if a pawn can complete this move will be stored in a separate class that will first return a board array to check if the move is possible. If enPassant is possible, the method will return both a move position and a remove piece position on the board. Allowing those two squares to be used for both moving and removing pieces on the board should the player choose this move.

### Update GUI

For each turn in the game the GUI for the chess board needs to be updated. In order to avoid slowdown or stuttering…

### Test game

In order to properly unit test the program, a simulated chess game will be required to show that all of the features implemented in the program are working. The creation and simulation of a new chess game will be required to demonstrate how effectively each aspect of the program will work.

### Check Checker

To detect whether a square is in Check a function has been made called checkChecker(). This function takes a Board and two coordinates as arguments. The board is the main board of the game instance, and the coordinates are the row and column coordinates of the square. The function returns a list of coordinates of pieces that are attacking the square, or an empty list if the square is not under attack.

The function gets the piece at the row and column coordinates, and iterates through each enemy piece, gets the possible moves for that piece, and if any of the possible moves lands the enemy piece on the square the function is checking, the coordinates of the enemy piece are added to a list which is then returned by the function once all the pieces have been iterated over.

### Checkmate Checker

To detect whether a king is in Checkmate a function has been made called checkMateChecker(). This function takes a Board and two coordinates as arguments. The board is the main board, and the coordinates are the row and col coordinates of the king. The function will return true if the king is in checkmate and false otherwise.

The function first gets the attacking color and the piece on the square that the function is checking. If the piece is not a King, then the function returns false. The function then gets a list of pieces attacking the King and creates a board to simulate the moves on. If the list is empty the function returns false as it cannot be checkmate.

If there is more than one attacker the king is in double check, and must move to escape checkmate, the function iterates over the king’s moves and if any of them put him out of check the function returns false as the king is not in checkmate, otherwise if none of them put him out of check it returns true.

If there is one attacker the king is in check and not double check, in which case the function iterates through the king’s possible moves and simulates them on the board, if the move gets the king out of check the function returns false, otherwise the simulation board is reset for the next move.

At this point none of the King’s moves can get him out of check, so the only way to escape checkmate is for a friendly piece to capture or block the attacker. The function iterates over each friendly piece and simulates a turn for each of the piece’s moves. If the move can block the attack or capture the attacking piece, the function returns false as it is not checkmate.

If the function has reached the end and has not returned, the king must be in checkmate, so the function returns true.

## Significant data structures

### Board Array

The board array data structure is a 2D array (matrix) of Square objects. Each square acts as a container for each part of the board which can contain any type of game piece or be empty. The board array does not handle any of the logic of the chess game, only acting as a ‘flat surface’ from which the chess game will be played on.

### Valid moves

Each piece on the board will contain their own arrayLists of the valid moves they can take, which can then be used by the makeMove class to help its calculations for where the piece can be moved to.

### Saving & loading

For saving and loading games, both for continuing running games as well as replaying previous game each step in the sequence of moves will be stored in an XML file. This is handled by the JAXB parser and the code in GameSaveManager. What data is saved and loaded and what data is excluded is controlled by JAXB annotations found in the major classes.

So that users can save game replays at the end of the game, each turn is saved to a temporary file. When the game ends and the user choses to save a replay, they specify a directory to save the replay in and the contents of the temporary file are copied into the directory and the temporary file is deleted. If the user does not want to save the replay the temporary file is deleted and not copied over.

REFERENCES

[1] Software Engineering Group Projects: General Documentation Standards. C.W. Loftus. SE.QA.05. 2.3 For Release

[2] Software Engineering Design Document (correct formatting needed here)

[3] Software Engineering GP02 Project User Interface Specification

DOCUMENT HISTORY

| *Version* | *Issue No.* | *Date* | *Changes made to document* | *Changed by* |
| --- | --- | --- | --- | --- |
| 0.1 | N/A | 27-02-2023 | N/A - original version | JAB153 |
| 0.2 | N/A | 07-03-2023 | Added interface descriptions | JAB153 |
| 0.3 | N/A | 08-03-2023 | Added UML sequence diagram and started significant algorithms | TPR3 |
| 0.4 | N/A | 09-03-2023 | Added interface descriptions | JAB153 |
| 0.5 | #43 | 11-03-2023 | Finished first UML sequence diagram as well added more descriptions to significant algorithms | TPR3 |
| 0.5.1 | N/A | 15-03-2023 | Added more significant classes | WIA14 |
| 0.5.2 | N/A | 16-03-2023 | Added method types | WIA14 |
| 0.5.3 | N/A | 19-03-2023 | Updated Significant Classes section | MJS36 |
| 0.5.4 | N/A | 19-03-2023 | Updated Game class interface description | MJS36 |
| 0.5.5 | N/A | 19-03-2023 | Updated Board class interface description | MJS36 |
| 0.5.6 | N/A | 19-03-2023 | Updated Square and Player class interface descriptions | MJS36 |
| 0.5.7 | N/A | 19-03-2023 | Updated Piece package interface description | MJS36 |
| 0.5.8 | N/A | 19-03-2023 | Updated Moves and Util package interface description | MJS36 |
| 0.5.9 | N/A | 19-03-2023 | Changed from boardArray:Square[][] to board:Board. | MJS36 |
| 0.6 | #43 | 19-03-2023 | Added all sequence diagrams | TPR3 |
| 0.6.1 | N/A | 19-03-2023 | Move a Piece algorithm description added | MJS36 |
| 0.6.2 | N/A | 20-03-2023 | Added detail to significant classes. Formatting changes to interface descriptions. Deleted test package | WIA14 |
| 0.6.4 | N/A | 20-03-2023 | Added Check and CheckMate functions to significant algorithms | MIB60 |
| 0.6.5 | N/A | 20-03-2023 | Added game save package to interface description, added more detail to saving and loading in significant data structures, added game save package to decomposition description | MIB60 |
| 0.7 | N/A | 20-03-2023 | Added UML class diagram, corrected file versions | WIA14 |
| 0.7.1 | N/A | 20-03-2023 | Fixed merge – saveReplay was missing in interface description | MIB60 |
| 0.7.2 | N/A | 20-03-2023 | Added more detail to Saving/Loading under significant data structures | MIB60 |
| 0.7.3 | N/A | 20-03-2023 | Fixed grammar and sentence structure issues in section 1 and 2. Updated GameSaveManager interface description to reflect same style as others | MJS36 |
| 0.7.4 | N/A | 20-03-2023 | Added more detail to Check and CheckMate in significant algorithms | MIB60 |
| 0.7.5 | #42 | 20-03-2023 | Added component diagram | JAB153 |
| 0.7.6 | N/A | 20-03-2023 | Castling Significant Algorithm updated | MJS36 |
| 0.7.7 | N/A | 20-03-2023 | Added reference to section 1. Corrected functional requirements mapped to modules | JAB153 |
| 0.7.8 | N/A | 20-03-2023 | Changed component diagram to a transparent version | MJS36 |
| 1.0 | N/A | 21-03-2023 | Submitted for review | WIA14 |
| 1.1 | N/A | 25-04-2023 | Added BoardGUI method | WIA14 |
| 1.1.1 | N/A | 26-04-2023 | Replaced all instances of x and y, with row and column | WIA14 |
|  |  |  |  |  |