Software Engineering Group Project - Maintenance Manual

Author(s): T.J Lewis [tyw1@aber.ac.uk]
Shaun Royle [shr27@aber.ac.uk]
Gwion Hughes [gwh18@aber.ac.uk]

Config Ref: Maintenance-GP9

Date: 13 February 2023

Version: 1.0

Status: Release

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this document

The purpose of this document is to provide the facilities for future maintainers of the project to understand its development process, properties, and its design, in order to make effective and efficient changes to the source code and other components of the project.

1.2 Scope

This document should be read by all team members, and any future developers or teams that intend to alter or improve upon the program's design or functioning in any way.

It is assumed that the reader is already aware of and familiar with the following documents:

- SE.QA.RS-CS22120 - Chess Tutor Requirements Specification [1]
- SE.QA.04 - User Interface Specification Standards [2]
- SE.QA.05 - Design Specification Standards [3]
- SE.QA.06 - Test Procedure Standards [4]
- SE.QA.08 - Operating Procedures and Configuration Management [5]
- SE.QA.09 - Java Coding Standards [6]
- UI-Spec-Docu-GP9 - UI Specification Document and Presentation [7]
- Design-Spec-GP9 - Design Specification [8]
- Test-Spec-GP9 - Test Specification [9]

1.3 Objectives

The document will address the following goals:

1. Describing the program structure and its functioning.
2. Detailing program properties and configuration details.
3. Discuss algorithms, resources, and data structures used.
4. Future alterations, improvements, changes, etc.
5. Maintaining the program effectively.

2. MANUAL

2.1. Project Configuration

Java JDK

The project was built and compiled using the **Amazon Corretto SDK (8.0.362)**.
This JDK was chosen due to its open source nature and wide availability, as well as being compatible with the target projects specified in SE.QA.RS-CS22120 [1]. It is also an LTS release, with a support schedule set to end in 2024. It is distributed under the GNU General Public License, meaning it is free to run, study, share, and modify.

Language Level

The project makes use of **language level 8**. When developing or making changes to the project, it is highly recommended that the language level be configured to be language level 17 or higher, to ensure compatibility.

JavaFX

To include graphics in the project, we opted to use **OpenJFX (19.0.2.x)**, imported from the Maven Central Repository. This JavaFX suite offers up to date functionality at the time of writing, and allows us to include a variety of graphical components in the project. The project uses the javafx-controls and javafx-fxml artifacts. It is distributed under the GNU General Public License 2, meaning it is free to run, study, share, and modify.

IntelliJ IDE

The project was developed using **JetBrains' IntelliJ iDEA IDE (2022.1.x)**, under the Free Classroom and Free Educational licenses, the former being provided to us by Aberystwyth University, as students.

Maven

The project makes extensive use of the **Maven 3 (3.8.x)** build system to compile and package the project, as well as including dependencies, and utilizing the various plugins that Maven offers. It uses the org.apache.maven.archetypes:maven-archetype-quickstart archetype, as it provides a simple and effective archetype to include the necessary dependencies and plugins required to develop, build, and package the project.

JUnit

The project was tested using the **JUnit (5.9.x)** unit testing framework and api. This framework provides a robust way of automating a large portion of the tests required to ensure functionality throughout the program. It also provides integration with the maven build system's lifecycle, allowing tests to be automatically executed when the program is compiled, packaged, or verified.

2.2. Program Description

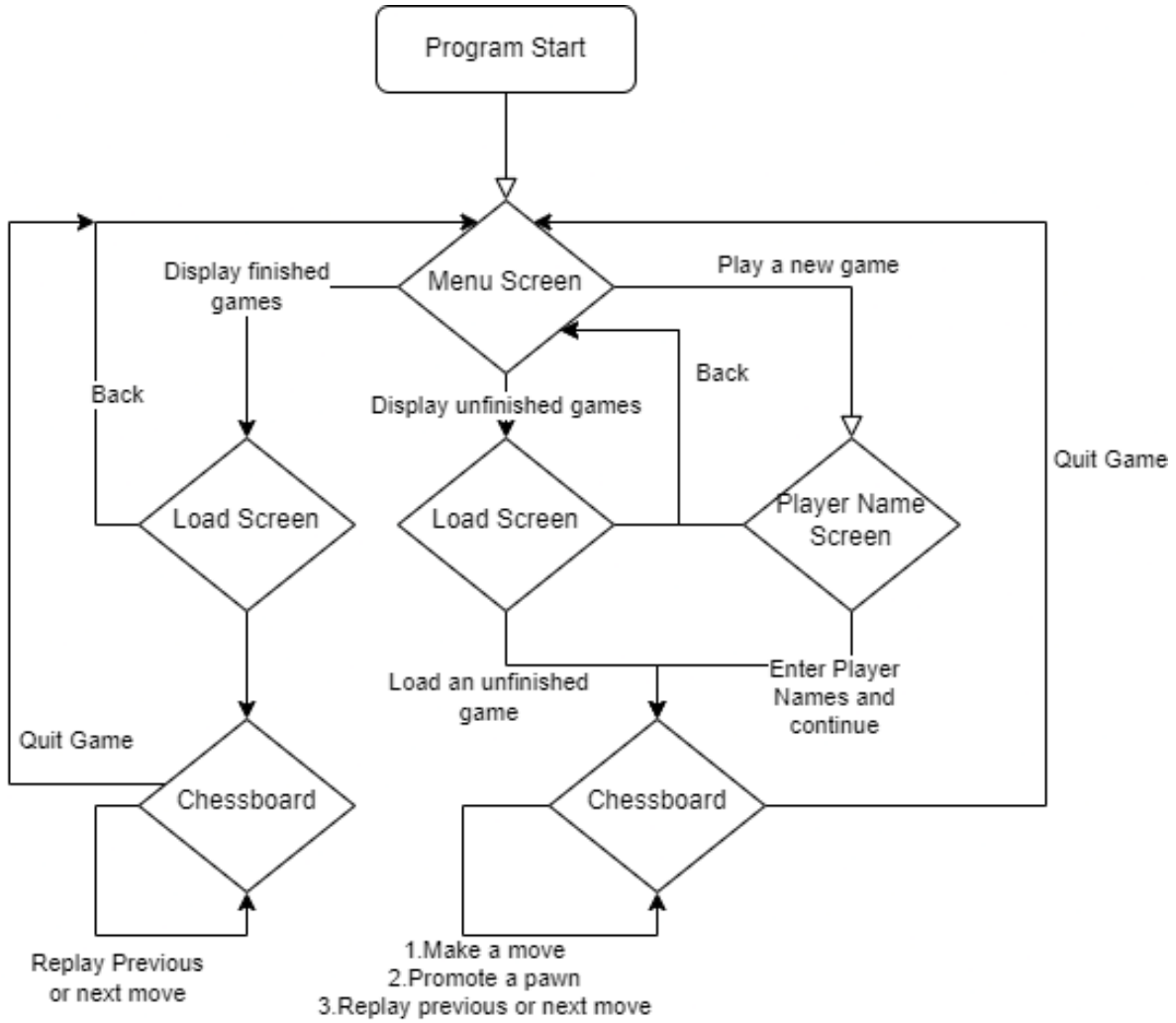
The aim of this program is to provide a solution that allows a user to play a game of chess, whilst being provided with a degree of assistance. It has two primary functional requirements, as detailed in SE.QA.RS-CS22120 [1];

1. To help users learn the valid moves in the game of chess, and to enforce those rules.
2. To allow two human players to use the game to play a game of chess.

Other additional and detailed functional requirements are detailed in SE.QA.RS-CS22120 [1]. The Design-Spec-GP9 document [8], section 2, describes how the program addresses those requirements with specific references to classes and functional requirements.

2.3. Program Structure

2.3.1 Flow of Control Diagram



2.3.2 Component Diagram

Component diagrams can be found in section 3, subsection 3.1, of the Design-Spec-GP9 document [8].

2.3.3 Sequence Diagrams

Sequence diagrams can be found in section 5, subsection 5.1, of the Design-Spec-GP9 document [8].

2.3.4 UML Class Diagrams

UML class diagrams can be found in section 5, subsection 5.3, of the Design-Spec-GP9 document [8].

2.4. Algorithms

Detailed descriptions for the following significant algorithms can be found in section 5, subsection 5.2, of the Design-Spec-GP9 document [8].

Read/Write Forsyth Edwards Notation strings.

The program makes use of the Forsyth Edwards board Notation. This notation allows the entire state of the board to be represented in a simple and effective string of uppercase and lowercase characters, as well as making use of delimiters such as “/”. The use of this allows the state of the game and board to be saved to a simple text file at any point. This helps to address the requirement of a crash-resistant project that can resume a game after a system failure/error, as well as being able to resume any previously saved games.

Calculating Legal Moves

The algorithm used to calculate the legal moves for the current players pieces. This is done before the player begins their turn, therefore *all possible moves are pre-calculated* before the player makes a piece selection.

The first step of the algorithm is to verify whether or not the player has been placed in check. This is done by first gathering the opponents potential movements, and comparing this to the current players' piece positions. If the king is under attack by any of the opponents players, they are considered to be in check.

The next step of the algorithm (providing the player is not in check) is to determine all of the possible moves that the player can make with each piece. The potential moves are then returned to be checked at a later time (when a player attempts to move a piece).

2.5. Data Structures

Forsyth Edwards Notation

The program makes use of a modified version of the Forsyth Edwards Notation [10]. The Forsyth Edwards Notation is a standardized method of storing and noting any chess board state as a string of characters and delimiters. The modifications made to this notation allow the program to recognise additional properties such as being able to determine whether a game has finished, which player has won, or whether it was a draw.

The Board

The board is represented by the Board.java class. Just as its real-life counterpart does, it contains all of the pieces that are in play, both white and black. A two-dimensional array is used to represent the positions of the board, as this best resembles the rank and file system of coordinates used in real-life chess games.

Coordinate

The Coordinate class represents a single set of coordinates relating to a position on the chess board, represented by int X and int Y. the coordinate class uses the method “public String getCoordinateAsChessNotation()” this translates the X and Y coordinate to a valid chess notation for example the coordinates X = 0, and Y = 5 will be translated to “a6”.

TileGraphicsLoader

The TileGraphicsLoader class handles the retrieval and storage of graphics relating to each of the pieces. The class utilizes a Java HashMap<Character, Image> in which the character is directly related to the FEN representation of a piece, and the image pointing to the image resource for that piece. The class then provides these resources to other components via the public fetchTilePieceGraphic() method.fetchTilePieceGraphic

Chessboard

The Chessboard class controls the graphical representation of the Chessboard. It differs from the Board class such that it is not logically aware of the state of the board, it only handles the drawing of graphics components on screen.

2.6. Files

Log Files

The Log class uses FEN strings in conjunction with standard text files to record the various stages of the games. This class handles all of the read/write functionality required, and therefore could easily be altered to utilize an alternative filetype or I/O system, should future development require it. These files are named “Log” files due to their purpose.

The log files are created when users initialize a new game, and are created outside of the project's root directory. This location was chosen over the standard Maven resources directory, as this would allow users to access log files easily without having to extract files from the final JAR file.

2.7. Interfaces

Interfacing between the front and back end is done through the game class this class is responsible for returning information to the front end in order for the front end to display valuable information, for example checkmate is calculated in the moveCalculator, which is then obtained by the game class and is returned to the interface class in order for the front end to graphically display this to the user.

2.8. Suggestions for Improvements

Implementing Modular Design

The project at present is centered around a number of high-control and powerful central classes that control the majority of the system's functionality. These are:

- Game.java
- Board.java
- MoveCalculator.java

These classes handle the majority of processing for the project. Future improvements could be the inclusion of a more modular control system, where additional controllers are introduced to create more granular control of the board and its pieces. This would also assist in making the codebase more human-readable and easier to understand.

Implementing Inheritance

The approach of using larger and more powerful classes, as mentioned under the previous heading, has led to many smaller classes having little to no control on the functioning of the project. This could be changed by introducing more inheritance focused components, such as a Piece abstract/interface class, that could then have subclasses of specific piece types (pawn, rook, king, etc.).

These subclasses could be configured to handle more functionality, such as determining their own available paths and moves, movement patterns, promotion opportunities, etc. This would create a more cohesive and object-oriented codebase capable of accepting further changes with ease.

Castling bug

A small bug was found post development which can be fixed, is that it allows the player to castle while they are in check in order to escape check, this move is not a legal move and will need to be fixed, a simple check before added a castling move to the list of the piece’s legal moves which checks if the player is in check can solve this quickly. Furthermore the player is able to castle through the threat of a piece which should be illegal and needs to be fixed for example if black’s light square bishop is on square c4 with a direct line of sight to square f1 the white king should never be able to castle king side because this square is in the king’s castling path and is being attacked by black’s light square bishop.

Support for Additional/Alternative File Types

The current logging system uses a combination of the FEN system and standard .txt files. The simplicity this offers is beneficial, however it can lead to hard to understand files and data that is harder to work with, test, and modify. The use of alternative data storage types such as XML, JSON, or Properties files, could greatly enhance the readability of the stored data, and allow it to be manipulated and managed in a more straightforward and easy to understand way. This would also make implementing saving and loading the players names easier.

Buffered I/O Interfaces

The program currently uses standard unbuffered I/O interfaces, such as the FileWriter. Future improvements could see the implementation of buffered interfaces, such as a BufferedWriter. This would offer a variety of benefits, such as increased performance, reduced I/O overhead, and more flexible output options.

Using FXML with JavaFX

The user interface currently utilizes the basic features made available by JavaFX, and offers a combination of logic and design control as part of the same components. The inclusion of FXML in the project could offer enhanced control over UI design features, and make updating or altering the UI straightforward. It can do this by separating the layout design from the application logic. This means a simpler system to maintain, as well as debugging and making improvements.

2.9. Making Changes

The project is split primarily into two separate packages, logic and graphics. As per the names, each contains parts relating to either the game logic or the game graphics. The contents of both packages are dependent on each other, so you must consider how the changes of one package may affect another.

Graphics Package

Any changes to FEN board notation or any moves from the FEN board notation as a way of storing the state of the board, would require revisions in the following classes. GraphicalController.java and PlayScreen.java. The GraphicalController passes the FEN notation from the backend to the playscreen. Inside the playscreen function, the FEN notation is split into an ArrayList of strings along the space delimiter. Rearranging the sections should be as trivial as tracking the function that handles those sections and reindexing them to their new locations. For instance, UpdateChessboard takes the first index of that arraylist. It should be noted the frontend does not handle anything but FEN strings to update the graphical interpretation of the chessboard. Therefore replacing the Forsyth Edwards Notation with some other notation or different data structure will be very challenging.

Any changes to the graphical display of the scenes should be easier. Search first for the constructor inside the class, then the method called to initialize the scene. In JavaFX, every instance of Button, HBox, VBox, is an instance of either an object you are seeing in the application window or an invisible container with alignment properties controlling the position of the object. If I wanted to add a new button to the start menu screen, I would search in StartScreen.java for where my buttons are declared, and which container they are added to. I would then instantiate a new button and add it to the container the other buttons are added to. If I wanted to make a button on the start screen that wasn't centered, or place my buttons in a more controlled manner, I would look into using a gridpane such as is used on the PlayScreen.java class to place all the objects on the screen. Making changes such as rearranging the scene containing the chessboard is as simple as locating where your

particular grievance is added to the gridpane object called “layout” and playing around with numbers until I got a good grasp of how these objects move.

If I wanted to update the sprites for the chess pieces, I would search for the resource folder and inside the images folder. Inside contains a selection of 50x50 pixel size pieces. First make sure that whatever sprites you wish to replace them with is 50x50 pixel size and that their backgrounds are transparent. Now replace each graphic you wish to swap with your own graphic but be careful to maintain the name.

If you want to move away from chess and implement some funky new pieces with different rules, make sure that any new sprites you have for these pieces are included on the HashMap inside the TileGraphicsLoader.java class. Remember the character that is its key as well and try to match it to whatever character will represent it in the FEN string passed from the backend.

Logic Package

Any changes to the structure of the Forsyth Edwards notation that we are using would have a large impact on the board class, as the board class is reliant on the FEN String being in a very specific format, and changes to it would have to be reflected in the code that parses the String and the code which builds the string back up after every move see the methods “initializeBoardState()”, and, “updateForsythEdwardsBoardNotation()”, one issue that could arise with the use of text files could be that the user would edit the FEN strings logged in the text files and essentially break the programs ability to correctly parse these strings (see 2.8). The FEN string being accurate is absolutely necessary for the front end to accurately show the state of the board.

In order for the moveCalculator class to work the method findLegalMovesForPlayer must be called with the “true” parameter then being called with “false” this is because when it is called with true it will construct the ArrayList of unsafe squares which it stores, and then when it is called with true it will use these unsafe squares in order to find out whether or not moves are legal using this constructed ArrayList on unsafe squares for the current player this is to determine whether or not moves will leave the player in check, this means that all the players moves are all calculated before a piece can be selected (see 2.8).

2.10. Physical Limitations

The chess tutor will run on most computers that have Java Runtime Environment 8.0.0_362 or later installed. The software is designed to run on faculty and IS desktops, its performance on other systems has not been tested and cannot be confirmed..

2.11. Rebuilding and Testing

In order to rebuild the project, it is required that the **Maven 3 (3.8.x)** build system is installed. You can then use the following lifecycle plugins to rebuild to project:

Phase	Description	Command
Clean	Removes all the compiled output files and other generated files created in the previous build in preparation for a new build.	mvn clean
Validate	Validates the project structure and that it includes all necessary information; dependencies, plugins, resources, etc.	mvn validate
Compile	Compiles the source code of the project and generates all required components.	mvn compile
Test	Executes any tests found in the testing directories, and displays the results of the testing phase in the terminal.	mvn test
Package	Gathers the compiled code, required resources, dependencies, and resources, and packages them into a JAR file.	mvn package
Verify	Runs integration tests on the created package to ensure that it works as intended.	mvn verify

To learn more about Maven and the lifecycle phases used, you can go to https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

3. REFERENCES

[1] SE.QA.RS-CS22120 - Chess Tutor Requirements Specification

[2] SE.QA.04 - User Interface Specification Standards

[3] SE.QA.05 - Design Specification Standards

[4] SE.QA.06 - Test Procedure Standards

[5] SE.QA.08 - Operating Procedures and Configuration Management

[6] SE.QA.09 - Java Coding Standards

[7] UI-Spec-Docu-GP9 - UI Specification Document and Presentation

[8] Design-Spec-GP9 - Design Specification

[9] Test-Spec-GP9 - Test Specification

[10] https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation

13. DOCUMENT HISTORY

Version	Issue No.	Date	Changes made to document	Changed by
0.1		01/05/23	Document created.	TYW1
0.2		10/05/23	Filled graphics section of 2.9	GWH18
0.3		10/05/23	Filled logic section of 2.9	SHR27
0.4		10/05/23	Added to improvements section	SHR27
1.0		11/05/23	Document Released	TYW1