\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SCHOOL OF ENGINEERING AND TECHNOLOGY**

**FINAL ASSESSMENT FOR THE BSC (HONS) INFORMATION TECHNOLOGY; BSC (HONS) COMPUTER SCIENCE; BACHELOR of SOFTWARE ENGINEERING (HONS)YEAR 2**

**ACADEMIC SESSION 2024; SEMESTER 3**

**PRG2104: OBJECT ORIENTED PROGRAMMING**

**Project DEADLINE: Week 14**

**INSTRUCTIONS TO CANDIDATES**

# This assignment will contribute 50% to your final grade.

* This is an individual assignment.

**IMPORTANT**

# The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

# Coursework submitted after the deadline will be awarded 0 marks

**Lecturer’s Remark** (Use additional sheet if required)

I Zar Nie Aung (Name) 21074380 std. ID received the assignment and read the comments



15 Jan 2025

(Signature/date)

**Academic Honesty Acknowledgement**

“I Zar Nie Aung student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment.”



15 Jan 2025 ….................................. (Student’s signature / Date)

1

**Table of Content**

[**1.0 Introduction 5**](#_h345qrw4kzb5)

[**2.0 Design and Development 7**](#_vuffz4ekq5zc)

[UML Class Diagram 7](#_7emf4z4xlyhv)

[UI Designs 11](#_6847l4sit50g)

[**3.0 Test Documentation 15**](#_ql5vot1qixgg)

[3.1 Entering Names for Game Activity 15](#_6e6rnja1sp1k)

[3.1.1 White Player and Black Player Filled 15](#_tnouc8zbqhgu)

[3.1.2 White Player Filled 16](#_6hvai5i4zris)

[3.1.3 Black Player Filled 17](#_a9n3fzg8w07y)

[3.1.4 Neither Players Filled 18](#_7z1f5ki1kw0j)

[3.2 Piece Event 19](#_ul7kfeakz0v)

[3.2.1 Pawn 19](#_91bs85jd8w7y)

[3.2.1.1 Pawn On Click Start 19](#_w7ovf0mzud62)

[3.3.1.2 Pawn On Move 19](#_6bft52o6j80w)

[3.2.1.3 Pawn On Click After First Move 20](#_p7nx8a4equn6)

[3.2.1.4 Pawn On Click Takeable 20](#_faaurpy8cbx0)

[3.2.1.5 Pawn On Take 20](#_ewj3ay2nnshr)

[3.2.2 Bishop 21](#_os04vyujxr6e)

[3.2.2.1 Bishop On Click 21](#_mx63w1ipjnbe)

[3.2.2.2 Bishop On Move and Takeable 21](#_2zgz7cuu0h76)

[3.2.2.3 Bishop On Take 22](#_611210ndzfi8)

[3.2.3 Knight 22](#_3ssf53h2991s)

[3.2.3.1 Knight On Click 22](#_skpygrmq0kci)

[3.2.3.2 Knight On Click, On Move, On Takeable 23](#_vuxcn7b1q2cl)

[3.2.3.3. Knight On Take 23](#_73dzxoxi962h)

[3.2.4 Rook 24](#_etmlemfb05jb)

[3.2.4.1 Rook On Click 24](#_y1qvvoj7apkm)

[3.2.4.2 Rook On Move, On Click, On Takeable 24](#_9eupkubr3dp6)

[3.2.4.3 Rook On Take, MoveTable On Scroll 25](#_to9f6kjo6z3h)

[3.2.5 Queen 25](#_h9by9medfde6)

[3.2.5.1 Queen On Click 25](#_4ukgam8p4nlz)

[3.2.5.2 Queen On Move 26](#_kr5jbvfl4itr)

[3.2.5.3 Queen On Move, On Click, On Takeable 26](#_k3x4ud22pvuv)

[3.2.5.4 Queen On Take 27](#_wc7j4m4i3d75)

[3.2.6 King 27](#_p6sqzb1cegpd)

[3.2.6.1 King On Click 27](#_mg8ykz4lfq2b)

[3.2.6.2 King On Move 28](#_w75a7qa5ow5k)

[3.2.6.3 King On Move, On Click, On Takeable 28](#_yso09z8mfr1l)

[3.2.6.4 King On Take 29](#_k3xt05jj96vq)

[3.3.Game-Over Event 29](#_bgqvga98104g)

[3.3.4.1 Black Won 29](#_wbvy7zedpmo3)

[3.3.4.2 White Won 30](#_cud9hnoj8gt)

[3.3.4.3 On Draw Event (Passive Play) 30](#_r49813nixp85)

[**4.0 Self-Reflection 31**](#_1ltqf699c7g7)

[**5.0 References 32**](#_sthil348k182)

# 

# 1.0 Introduction

Long Live the King (LLtK) is an adaptation to the standard chess game where the end goal of the is to take the opponent’s king piece. LLtK is played between 2 players on a more minimal chess engine. Players are not stopped during check conditions and are not prevented from making moves that might lead to one, especially if the move benefits their position. This leads to a more unique feel of chess, where players are to solely rely on their knowledge of chess to keep their king on the board, whilst also adapting to new strategies in order to knock their opponent’s king off the board unnoticed. With this new end goal, a game of LLtK is then played with the following set of rules:

* **Players are free to make any standard chess move except**
  + Pawn promotions - The premise of the game is to ‘kill’ the other king, making pawn promotions overpowered and highly valued early game. This also makes all existing pieces more valuable as they shall not be replaced later in game, forcing players to make due with what they currently have to win the game.
  + Castling - The game is built upon a limited game engine with the aim of forcing players into a more aggressive play style. Any defensive move like castling is not supported.
  + En passant - Similar to the above reason, the game engine was intentionally under developed, preventing special moves like en passant to be played. This way not only are players forced to be more aggressive, they are also forced to learn a new style of chess.
  + When making a check move, the player are advised to *not* inform their opponent - The premise of the game is to take the other king. Informing of the check, though not against the rules, would certainly undermine the player’s efforts at winning the game.
* **Players are to make an aggressive at least once in every 5 rounds**
  + The game offers minimal support to protect players from check conditions, forcing many to scan the board regularly. The level of pressure may lead to players intentionally making more defensive moves rather than offensive ones. To prevent this, all players must take one enemy piece at least once every 5 rounds, or else a draw will occur. Moreover, this can also be exploited by a player with lesser material, forcing the player with superior material to find a way to get the enemy king or another enemy piece as soon as possible. This approach does not stray far from standard chess, as when a player’s conditions are unfavorable, they can still avoid losing by forcing a draw condition.
* **The King must fall**
  + Through any means necessary, the player must be able to take the opponent player’s king.

In consequence to the above rules, LLtK is built surrounding the following functional requirements.

|  |  |
| --- | --- |
| User shall be able to move a piece | For all pieces the player can move, the game should display all the possible legal moves said piece can make.  The user shall be then able to select a tile on the board for the said piece to move to. |
| User shall be able to take a piece | If an enemy piece exists in range of the selected piece’s moves, the user shall be able to take the piece. |
| User shall be able to quit the game at any point | A game of chess can be relatively long and uninteresting. Because of this, the game view shall contain a quit button where users can leave the game easily. |
| User shall be able to keep track of the game progression | As per standard chess games, such as Chess.com, most players are able to see all the moves that have occurred throughout the game in order to keep track of game events and learn their opponent’s strategy.   To meet this requirement, the game view shall display these moves. |
| User shall be able to win, lose | As per the rules of the game, the user shall be able to win or lose depending on if they took the opponent’s king piece, or if their king piece was taken.  Moreover, if both players played passively for 5 rounds in a row, both players will end up in a draw. |
| User shall be able to display their name during the game play. | If a user is willing to put their name under a respective team, their name shall be displayed on the respective card of the game view. |

# 

# 2.0 Design and Development

LLtK is designed and developed using Scala 3 as the codebase following the **M**odel-**V**iew-**C**ontroller software model.. The User-Interfaces (Views) were designed using JavaFX components, and were controlled using controller classes built with the ScalaFX library.

## UML Class Diagram

Below is the UML-class diagram of LLtK. Much more information of their implementation can be found in the source code as scala comments.

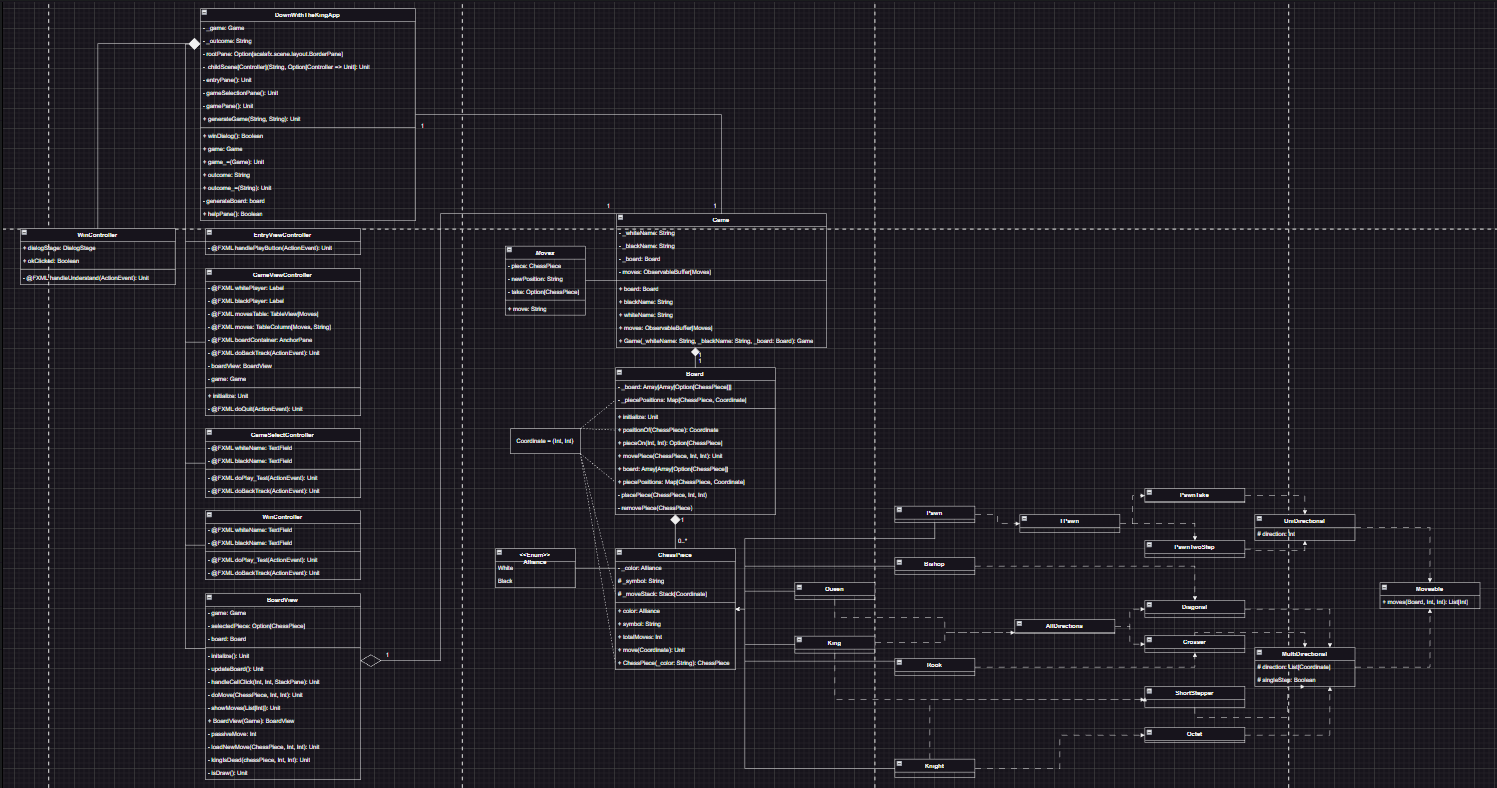


Figure 1 - Full UML Class Diagram of LLtK

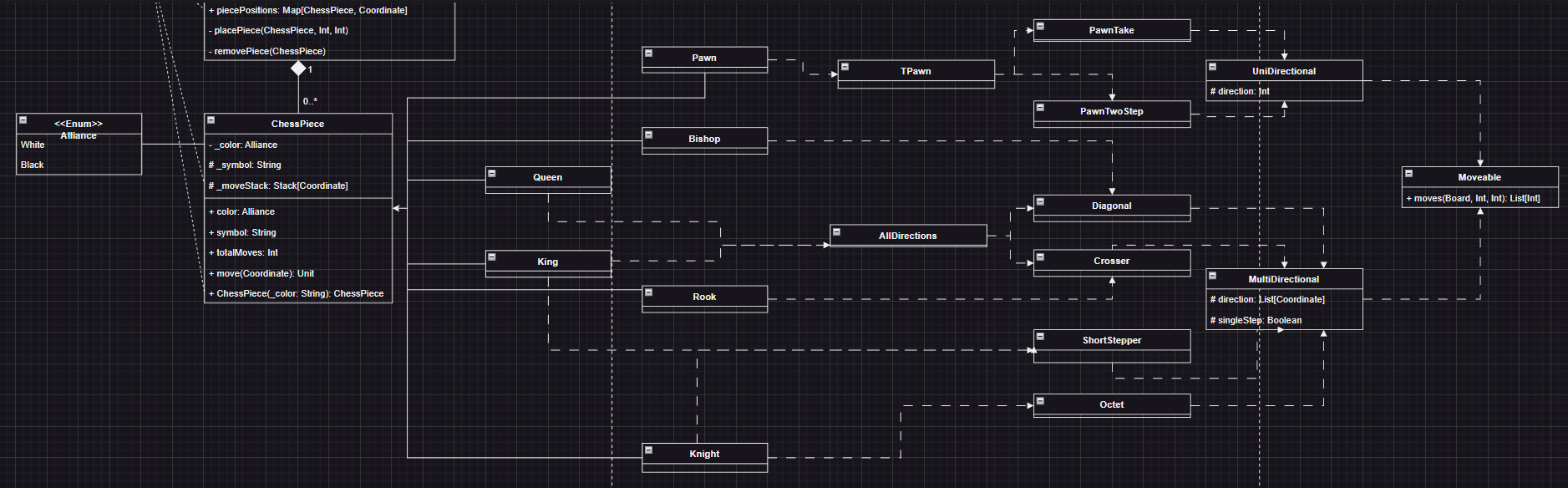


Figure 2 - ChessPieces and Traits

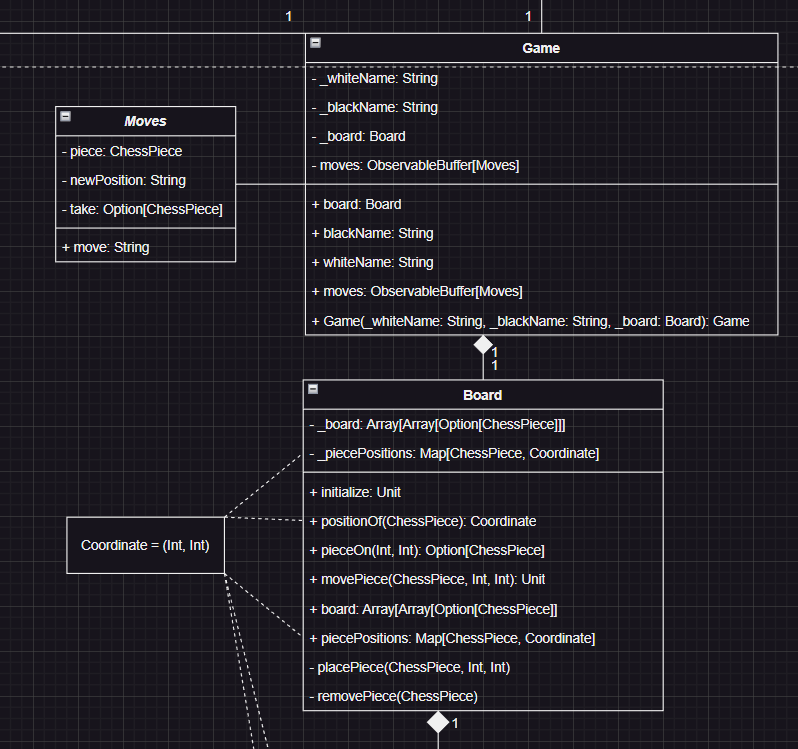


Figure 3 - Board, Game, and *Moves* (case class) classes.

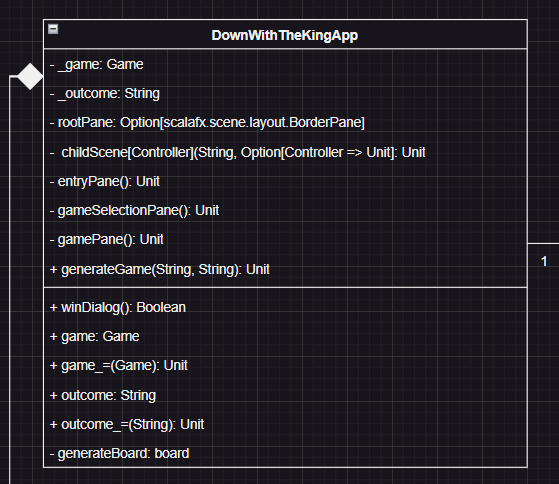


Figure 4 - Main App Object LongLiveTheKing (LLtK)

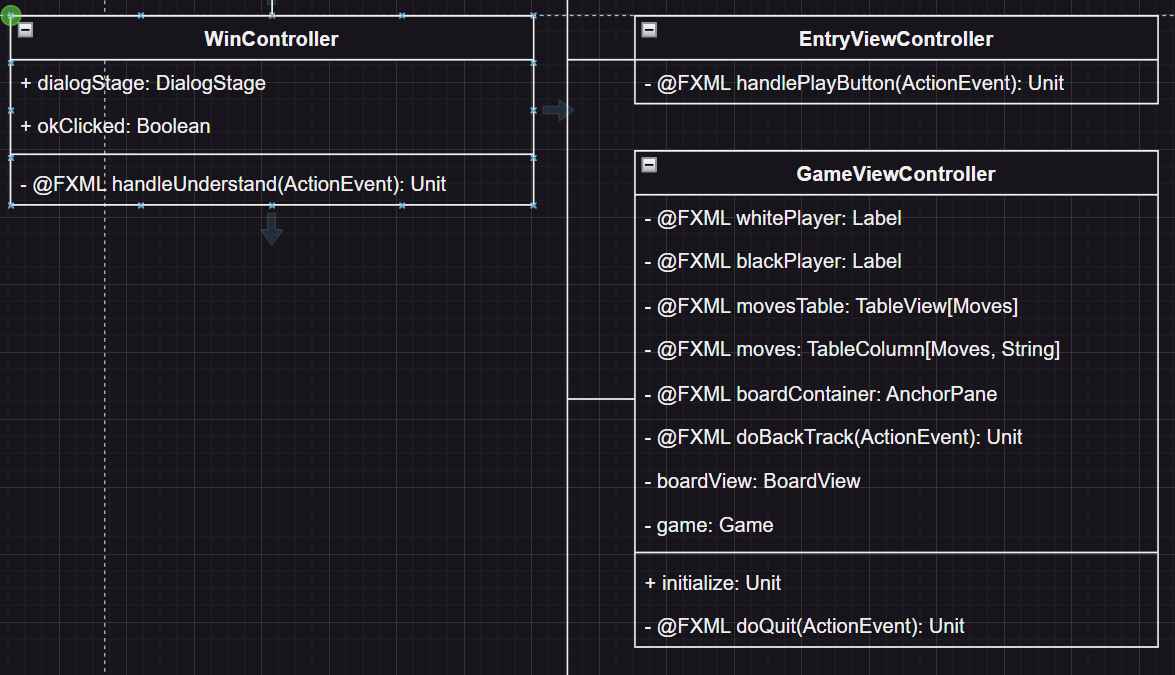


Figure 5A - Children Scenes of LLtK

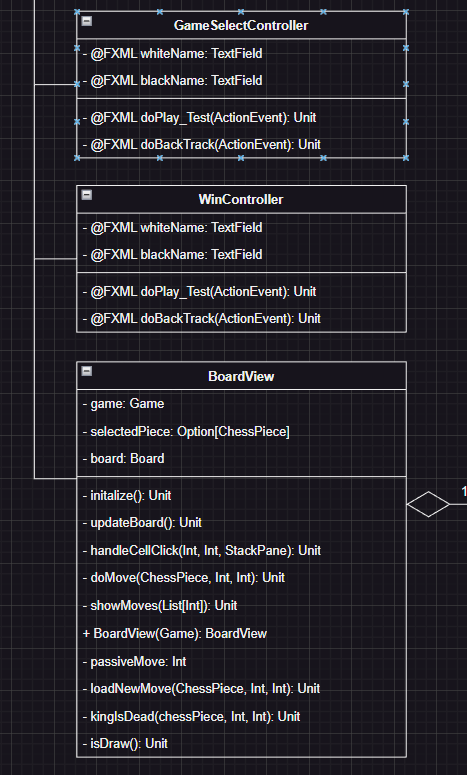


Figure 5B - Children Scenes of LLtK

## 

## UI Designs

Provided below are the UI snippets that ChessTutor features. These UIs are built using SceneBuilder, a powerful UI building tool for JavaFX FXML GUIs.

RootLayout.fxml is the master view of the application. It’s responsible for containing common functions that the entire application will depend upon, and also contains key-bind methods to all game related functions, such as create a new game, quit game, and so on, as well as include assistance information. The root node is **BorderPane**, and in it’s center other main scenes will be loaded into it.

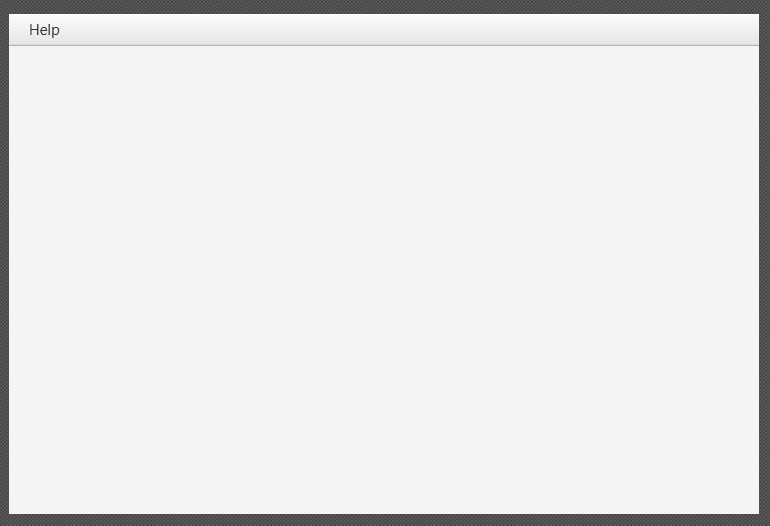


Figure 6 - RootLayout.fxml

EntryView.fxml is the main entry point view of the code. It is responsible for providing user’s a user-friendly navigation to two core components of the application, GameHistory.fxml (Fig 4) and GameSelect.fxml (Fig 5). This scene was built following standard HCI protocols to offer a user-friendly experience when navigating across different scenes.The root node is **AnchorPane**.

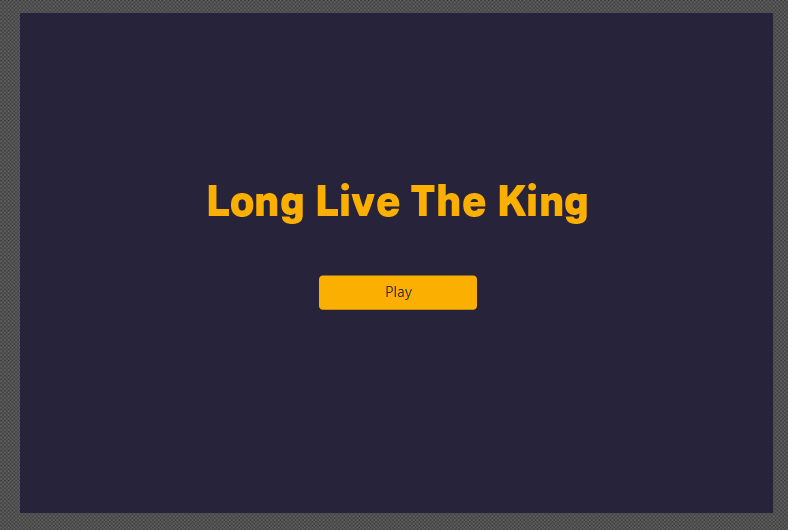


Figure 7 - EntryView.fxml

GameSelect.fxml is responsible for building the new game instance object that will be used by the GameView.fxml (Fig 6). On the top of the scene exists a back button bar to EntryView.fxml (Fig 1). In the center of the screen exists a 2 text fields, where players can enter their names, and a button below them to start the game event. The root node is **AnchorPane**.

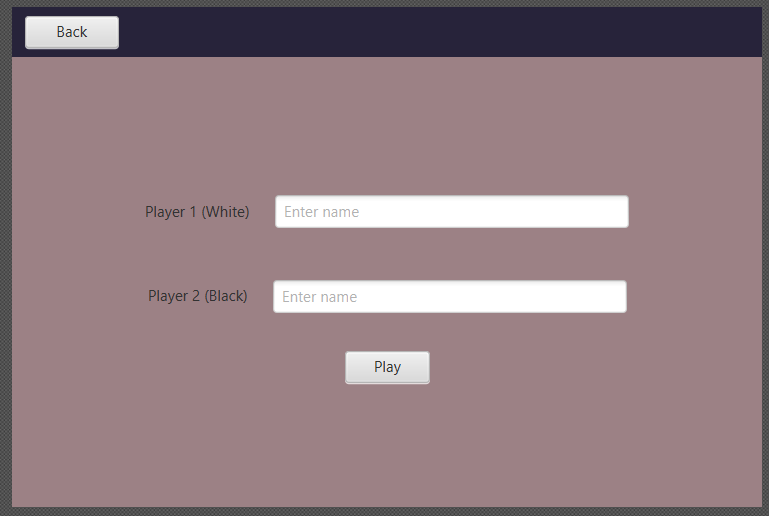


Figure 5 - GameSelect.fxml

GameView.fxml is responsible for loading game instance object information, and controlling the game logic. It is built using nested **SplitPanes**. On the left split pane, exists a horizontal **SplitPane**. On the top is a **TableView** containing the list of moves that has occurred throughout the game. On the bottom is the quit button to allow players to stop playing the game. On right split pane is a **AnchorPane** nested with a **VBox**. The top of the Vbox has an **Hbox** with nested VBox elements to store both the players’ names and their respective alliance. Beneath the Hboxes is a single AnchorPane where the BoardView instance will be loaded onto. The root node is **AnchorPane**.

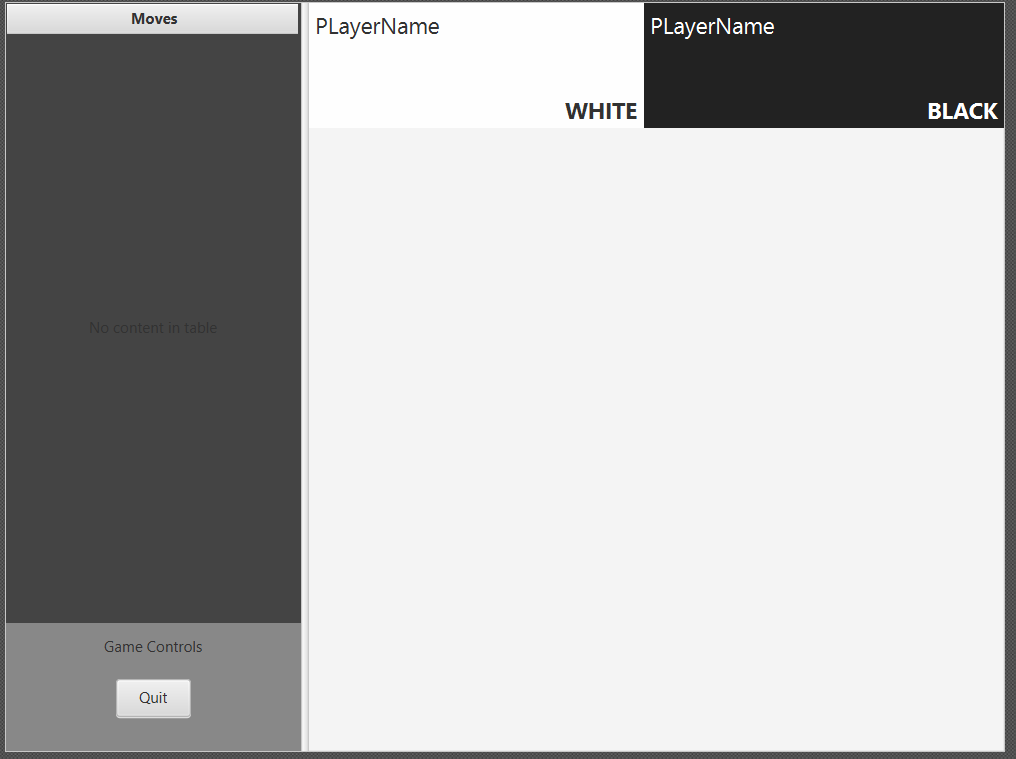


Figure 5 - GameView..fxml

WinView.fxml is responsible for displaying the game outcome message. It is built with nested **VBox** and **HBox** elements as they offer more flexible orientation. The root node is **AnchorPane**.

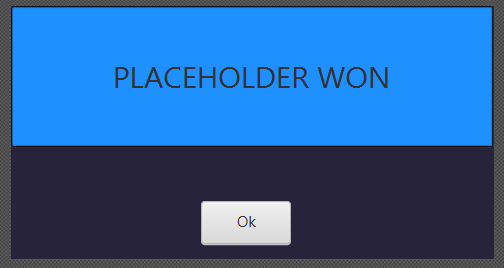


Figure 6 - WinView.fxml

RulesView.fxml is a pop up pane that can be accessed on the RootLayout.fxml under the help button or through the shortcut `*Ctrl+H*`. It contains all the basic information of LLtK, the rules the players should follow during a game event, and how to win. This approach was taken as this allows players to easily access the rules view regardless of their positon on the main view.

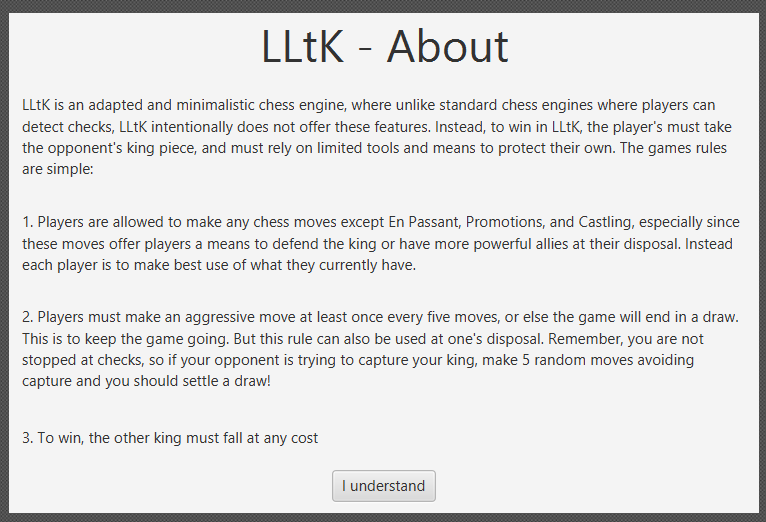


Figure 7 - RulesView.fxml

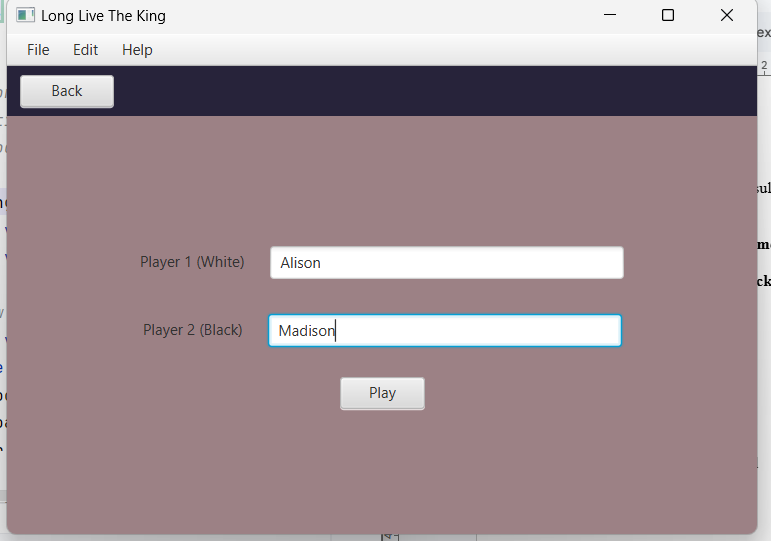
# 

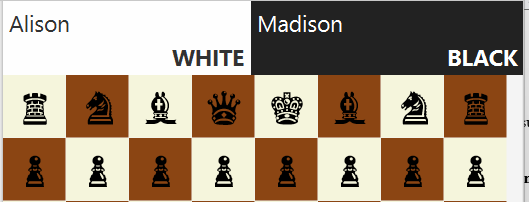
# 3.0 Test Documentation

In this section, all the test results of the proposed functional requirements of LLtK will be demonstrated here.

## 3.1 Entering Names for Game Activity

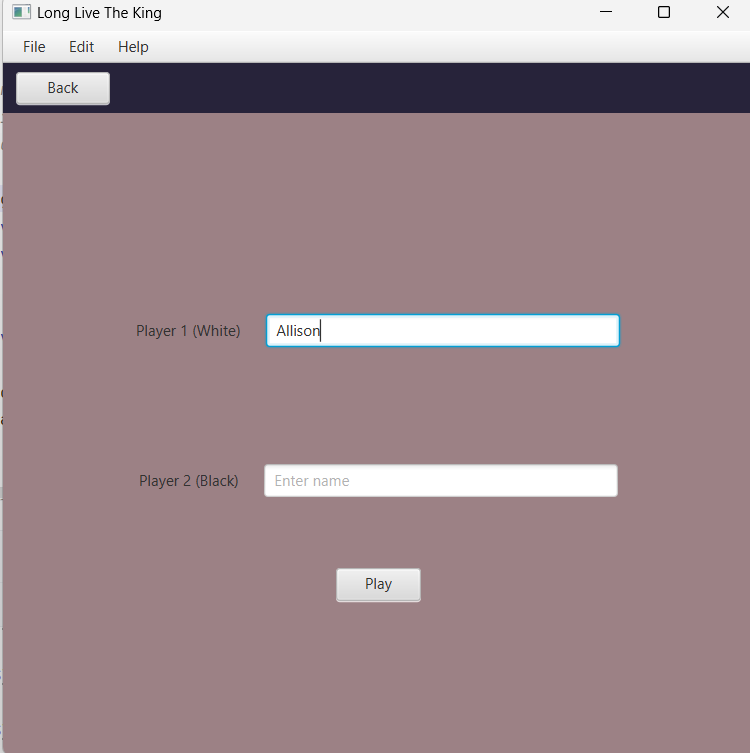
### 3.1.1 White Player and Black Player Filled

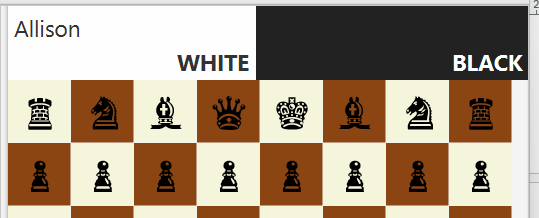




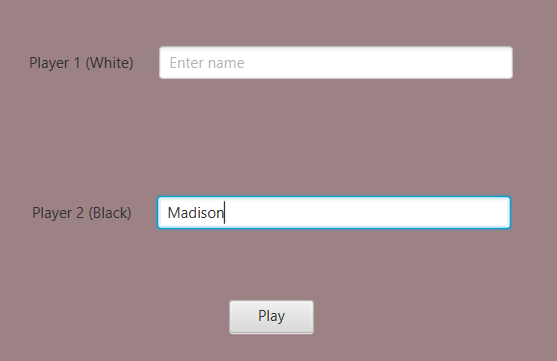
### 

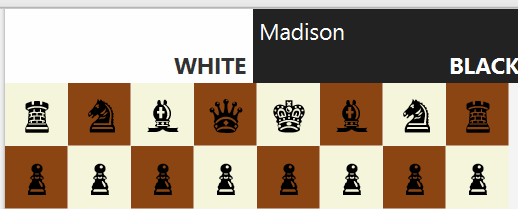
### 3.1.2 White Player Filled





### 3.1.3 Black Player Filled





### 3.1.4 Neither Players Filled

## 

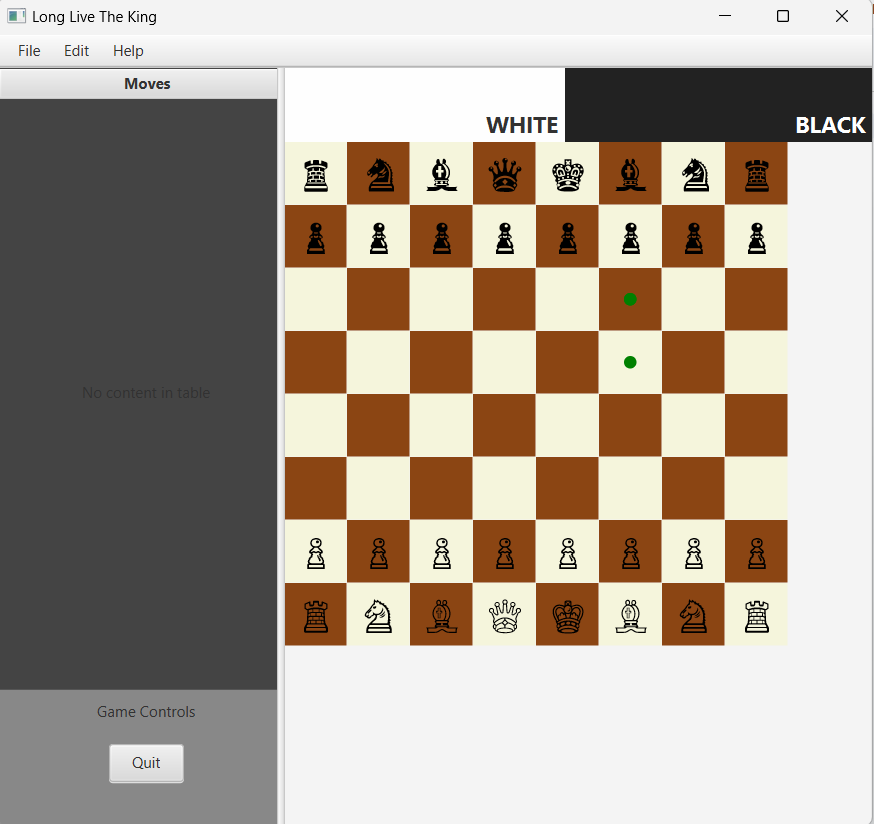
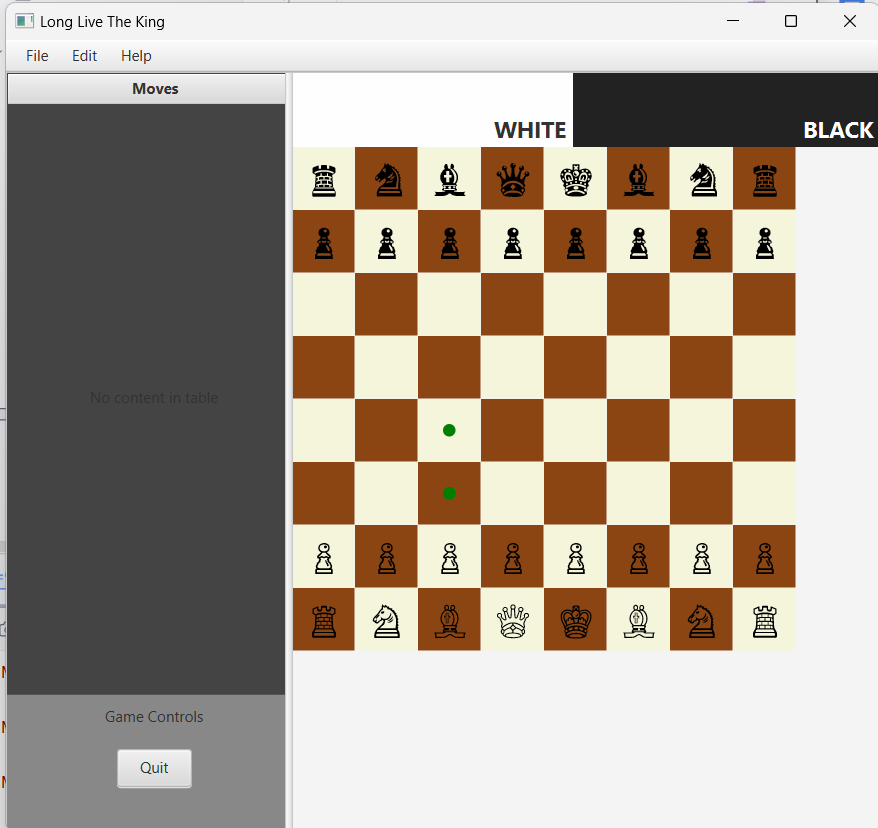


## 

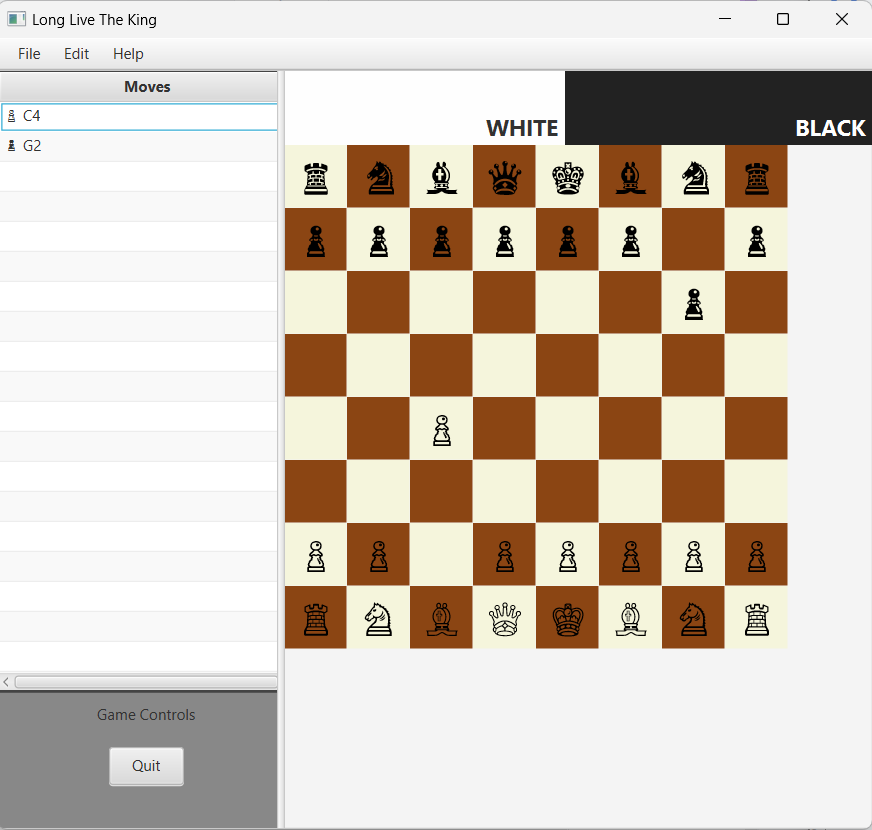
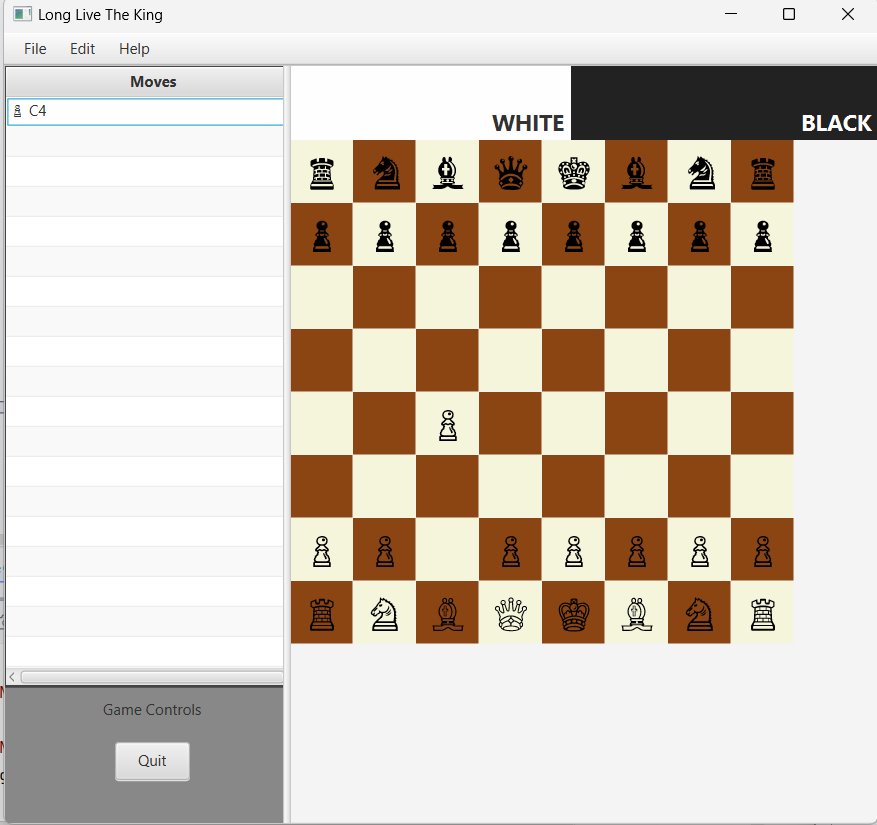
## 3.2 Piece Event

### 3.2.1 Pawn

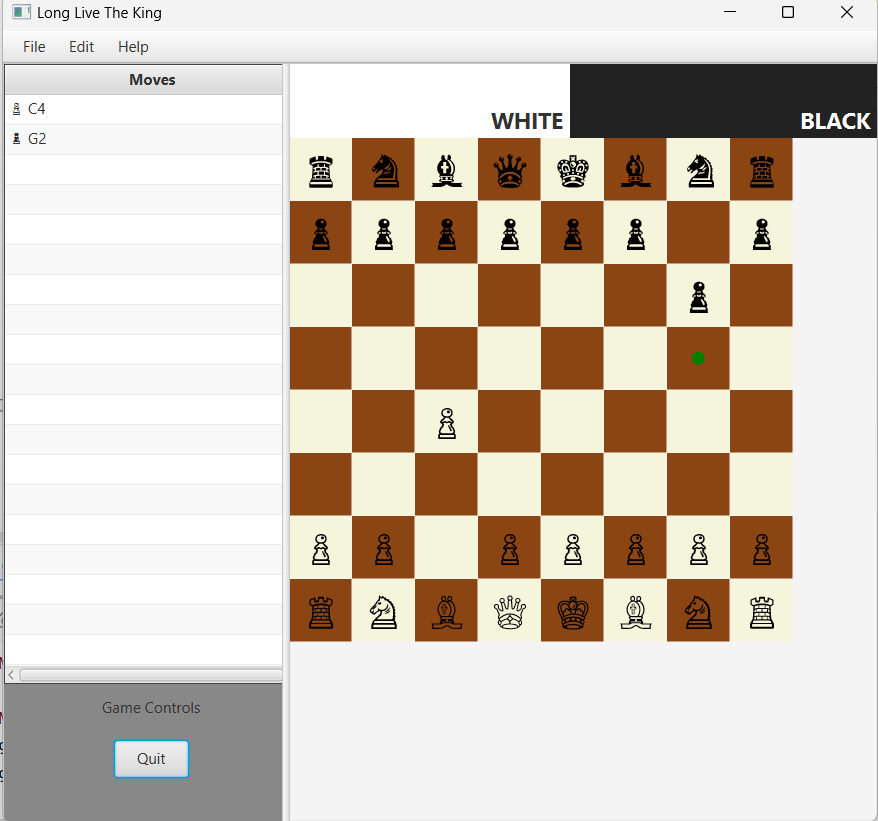
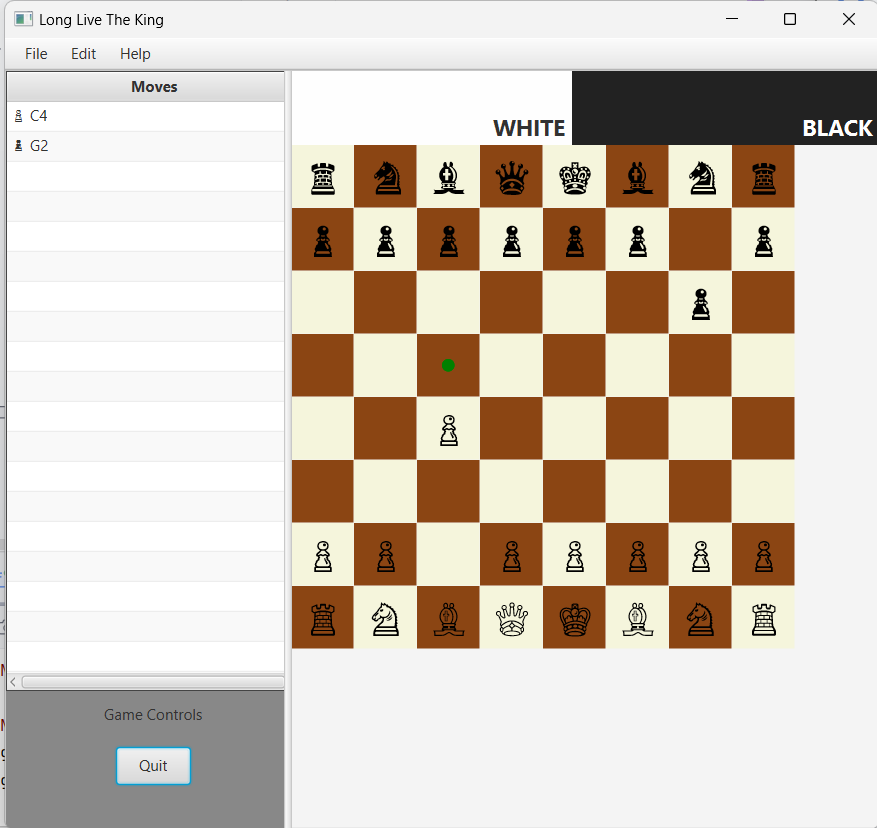
#### 3.2.1.1 Pawn On Click Start



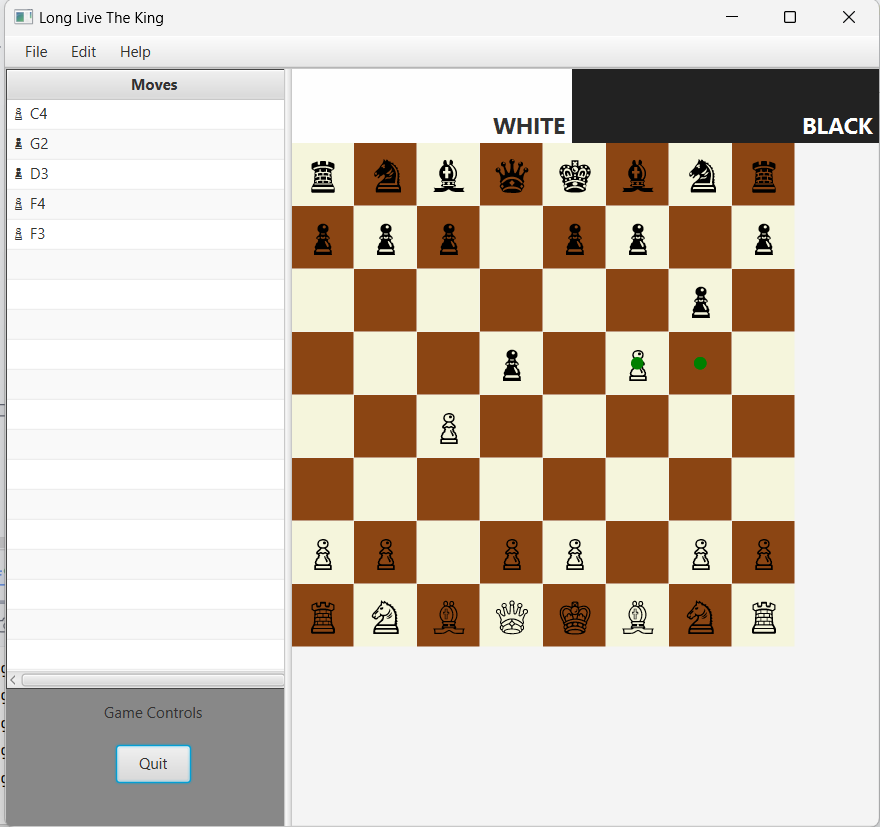
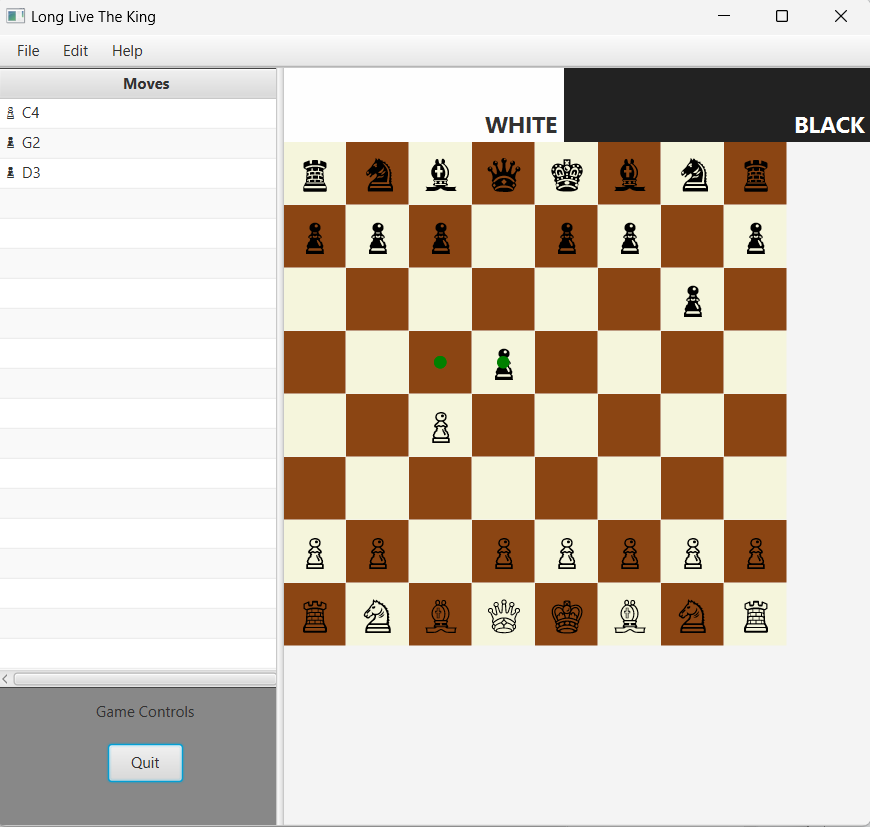
#### 3.3.1.2 Pawn On Move



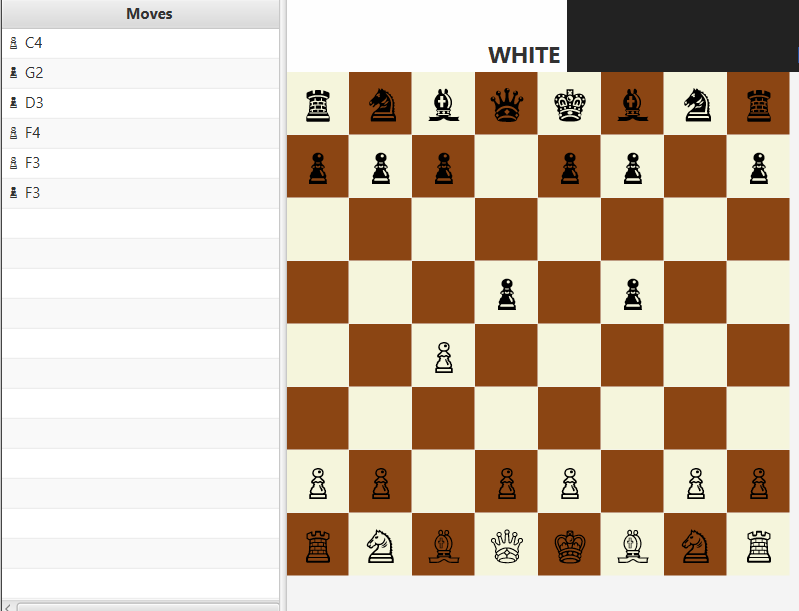
#### 3.2.1.3 Pawn On Click After First Move



#### 3.2.1.4 Pawn On Click Takeable

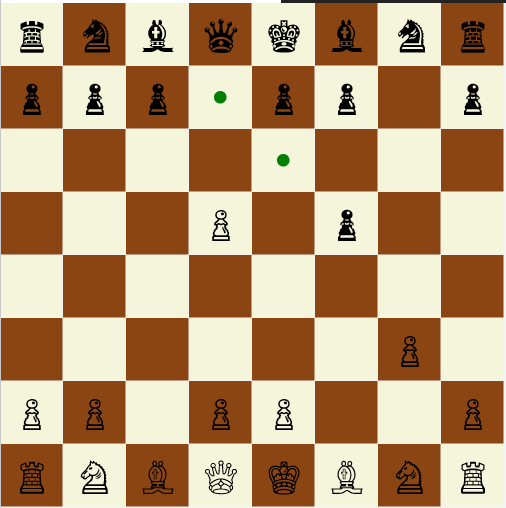


#### 3.2.1.5 Pawn On Take

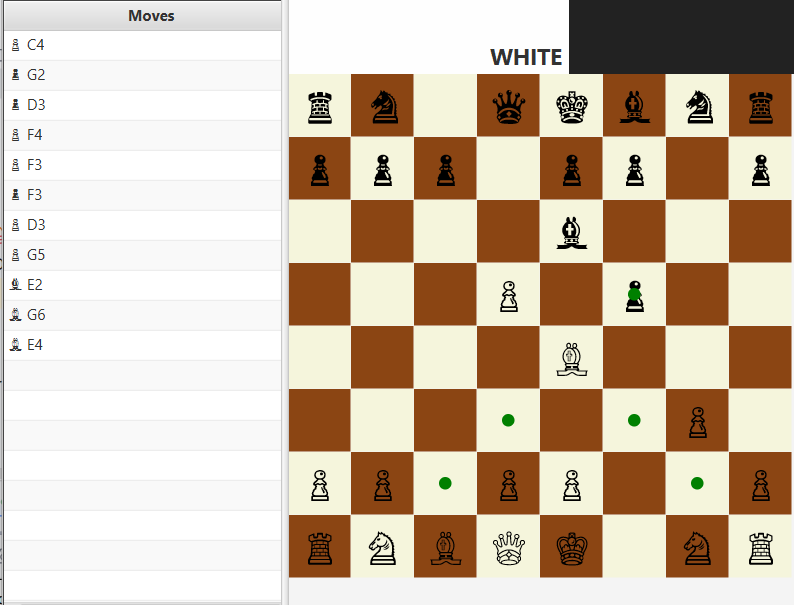


### 3.2.2 Bishop

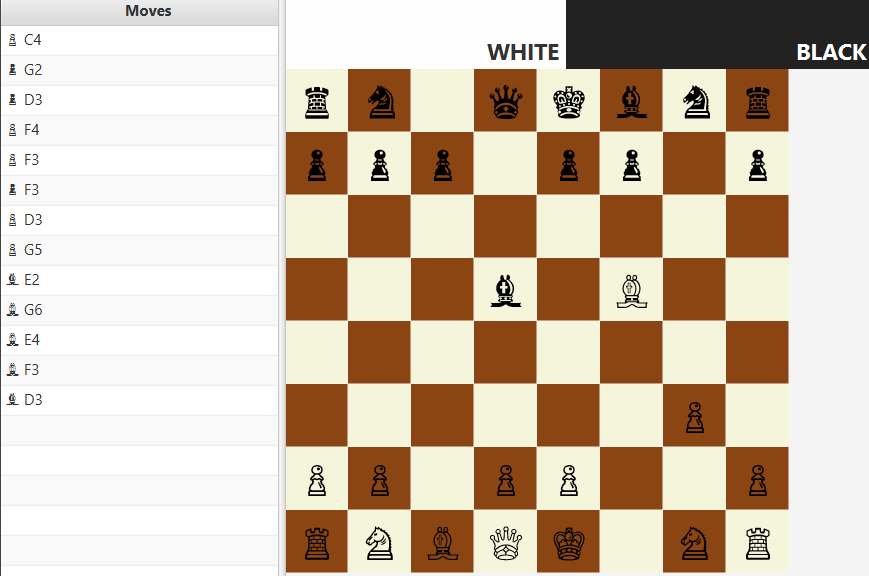
#### 3.2.2.1 Bishop On Click



#### 3.2.2.2 Bishop On Move and Takeable

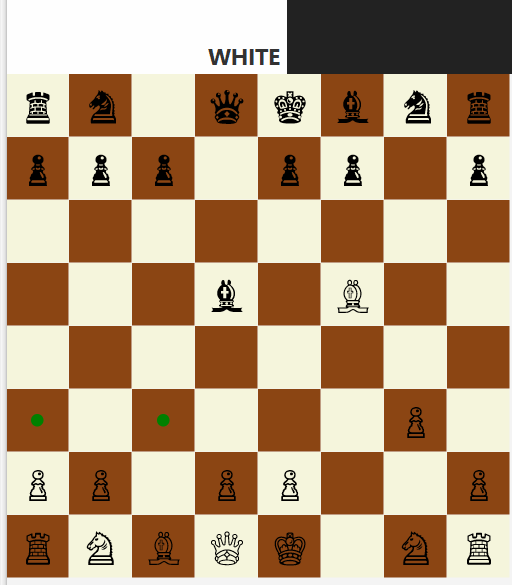


#### 3.2.2.3 Bishop On Take

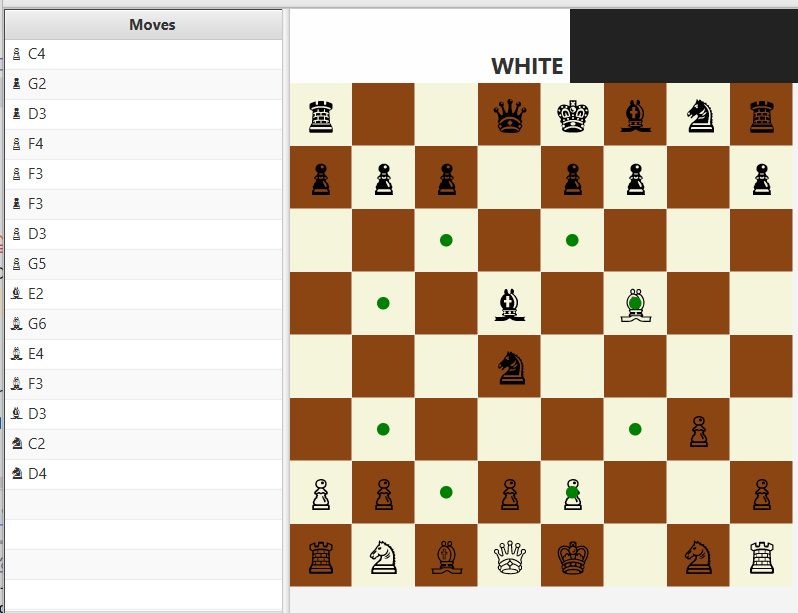


### 3.2.3 Knight

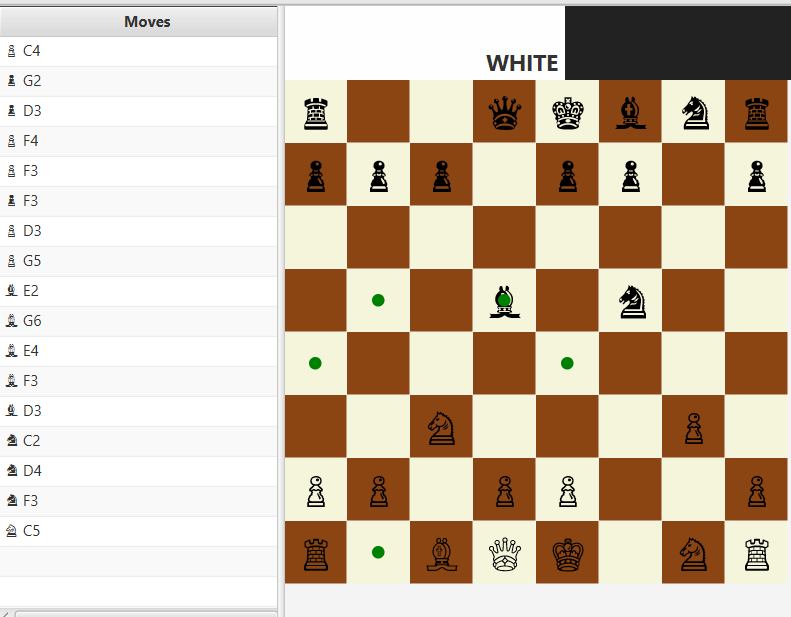
#### 3.2.3.1 Knight On Click



#### 3.2.3.2 Knight On Click, On Move, On Takeable

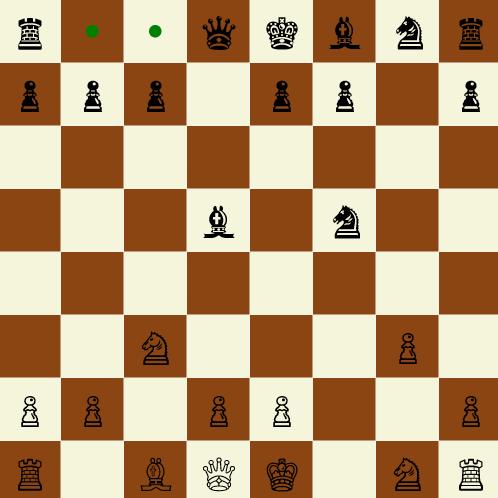


#### 3.2.3.3. Knight On Take

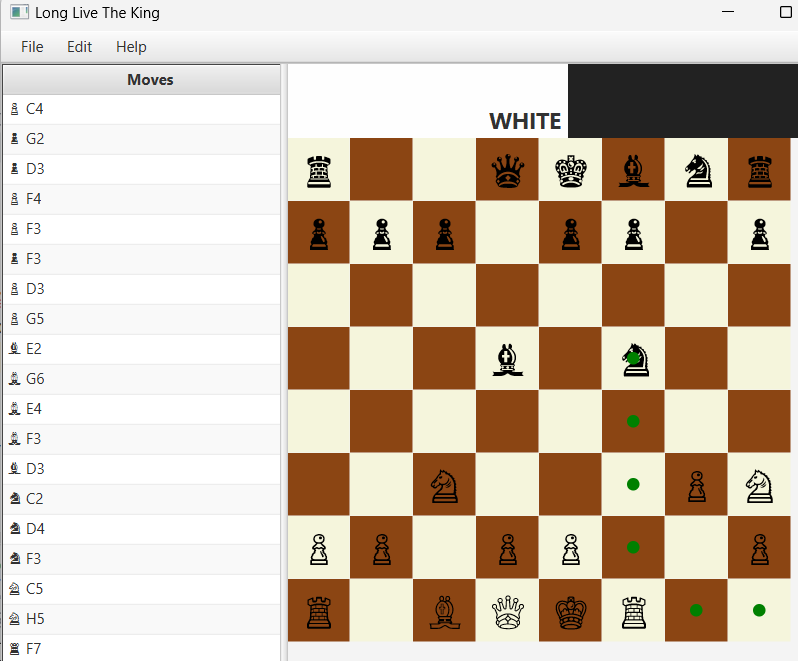


### 3.2.4 Rook

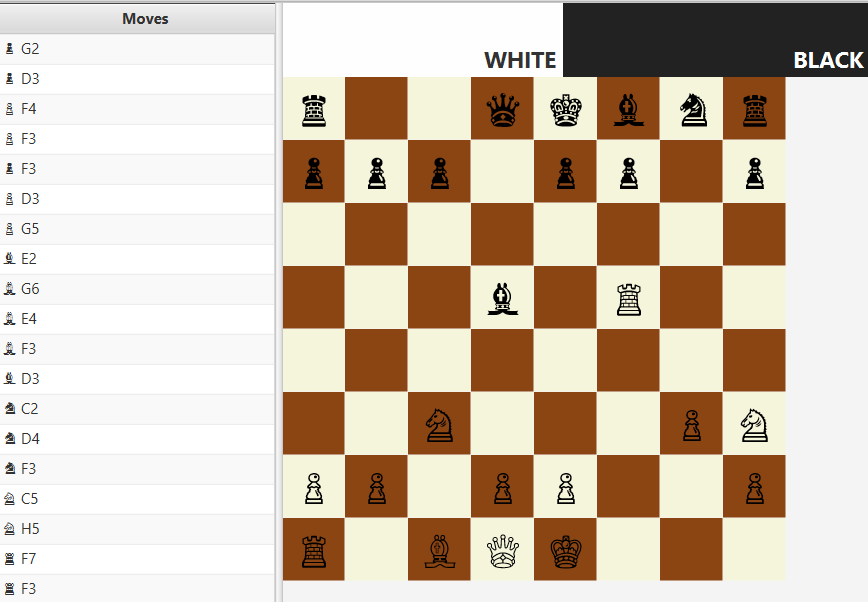
#### 3.2.4.1 Rook On Click



#### 3.2.4.2 Rook On Move, On Click, On Takeable

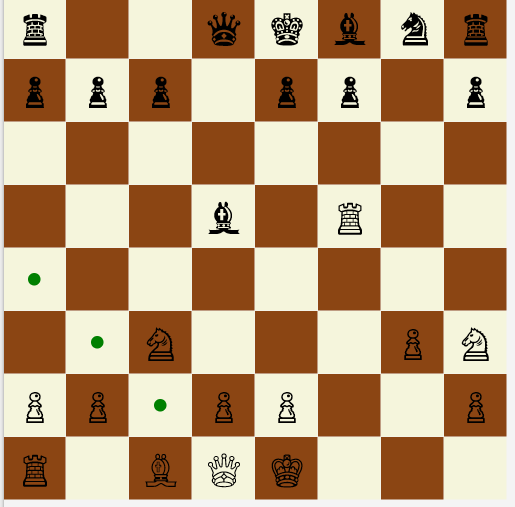


#### 3.2.4.3 Rook On Take, MoveTable On Scroll



### 3.2.5 Queen

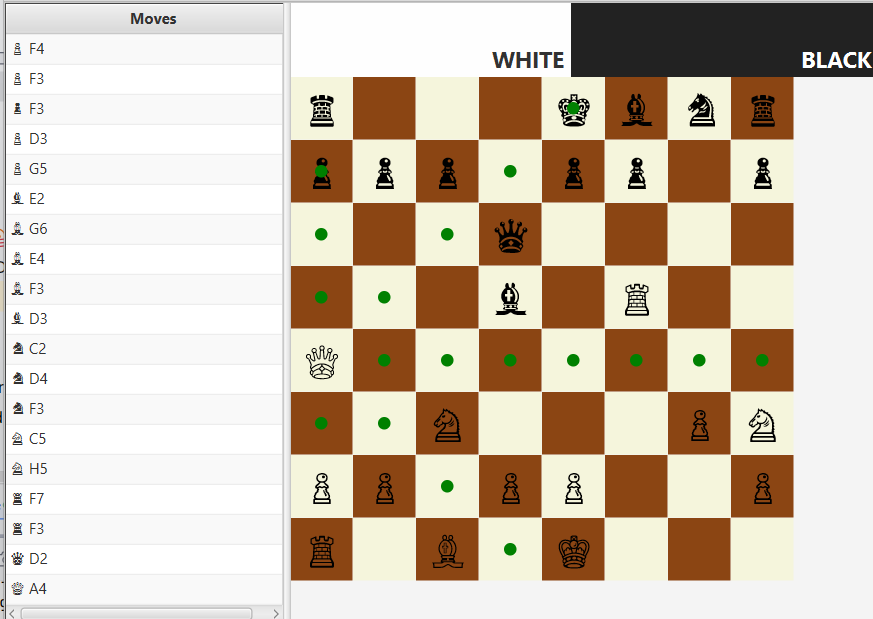
#### 3.2.5.1 Queen On Click



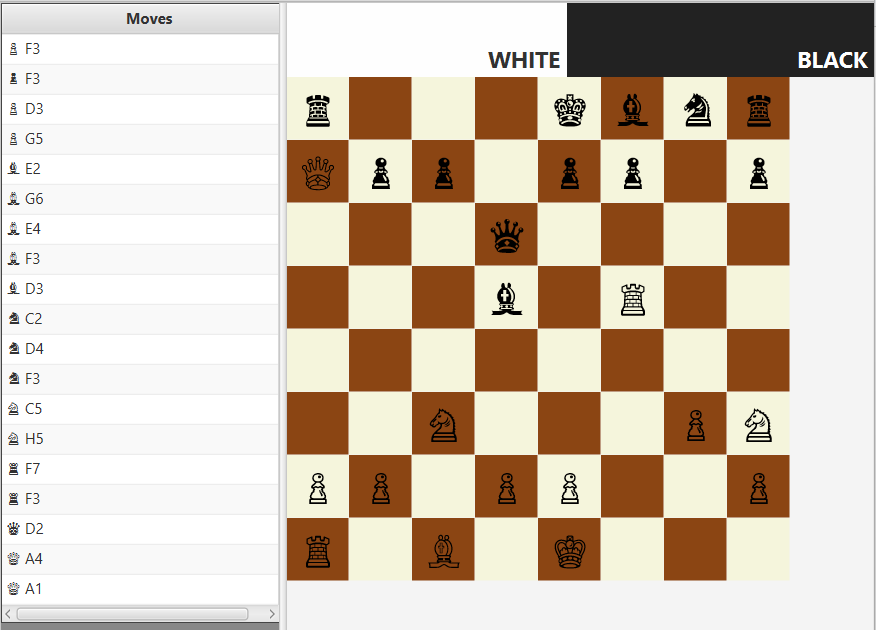
#### 3.2.5.2 Queen On Move



#### 3.2.5.3 Queen On Move, On Click, On Takeable

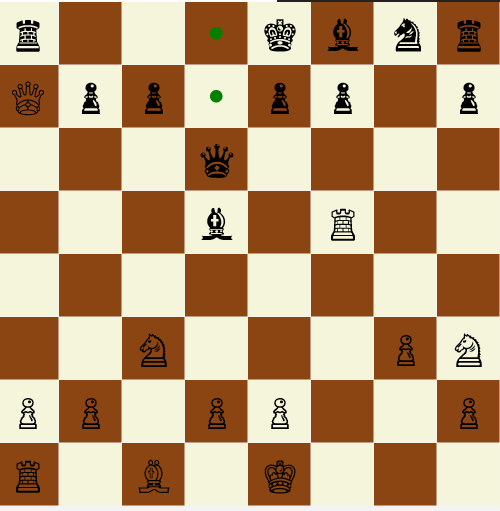


#### 3.2.5.4 Queen On Take

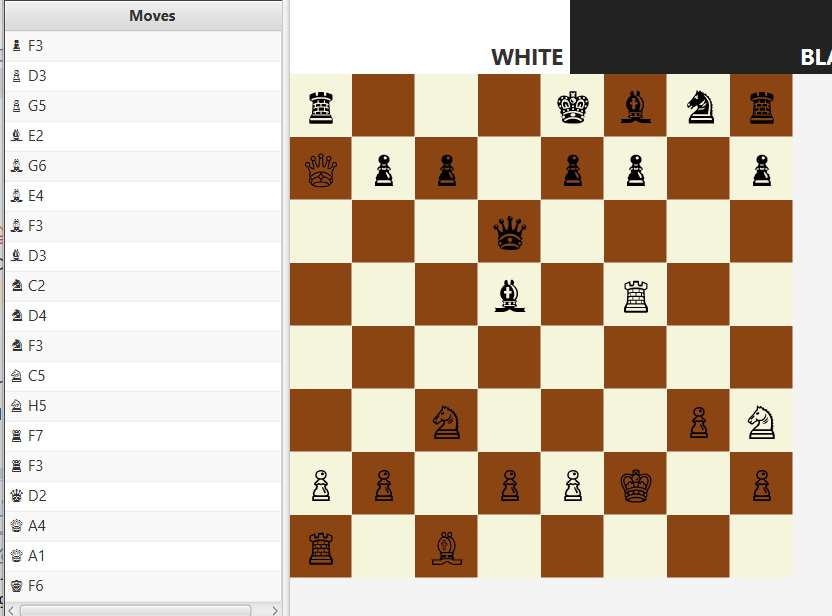


### 3.2.6 King

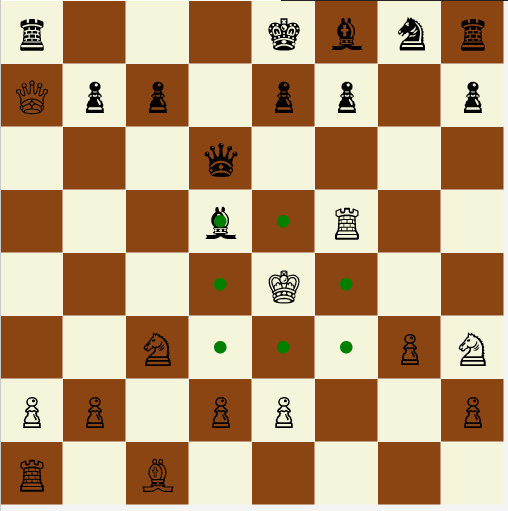
#### 3.2.6.1 King On Click



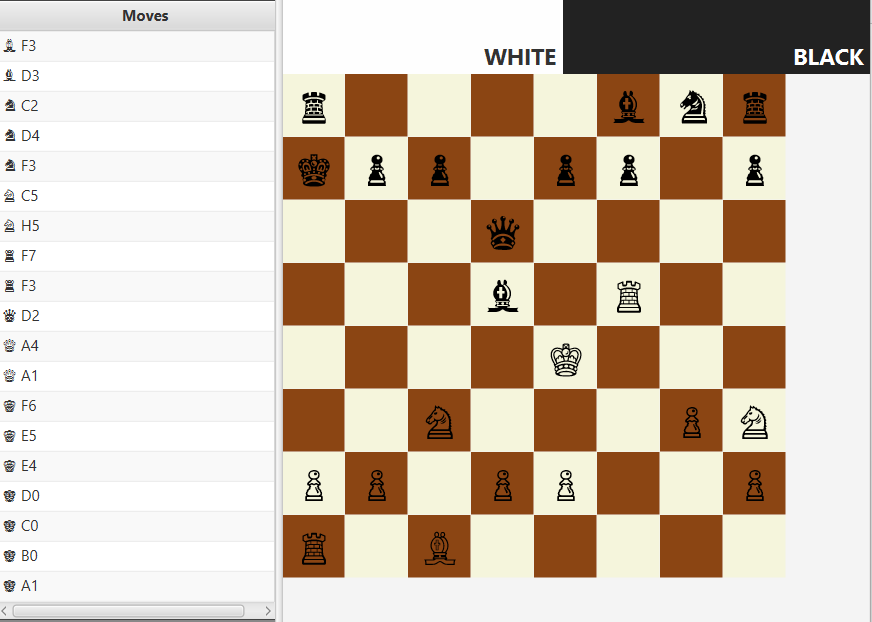
#### 3.2.6.2 King On Move



#### 3.2.6.3 King On Move, On Click, On Takeable

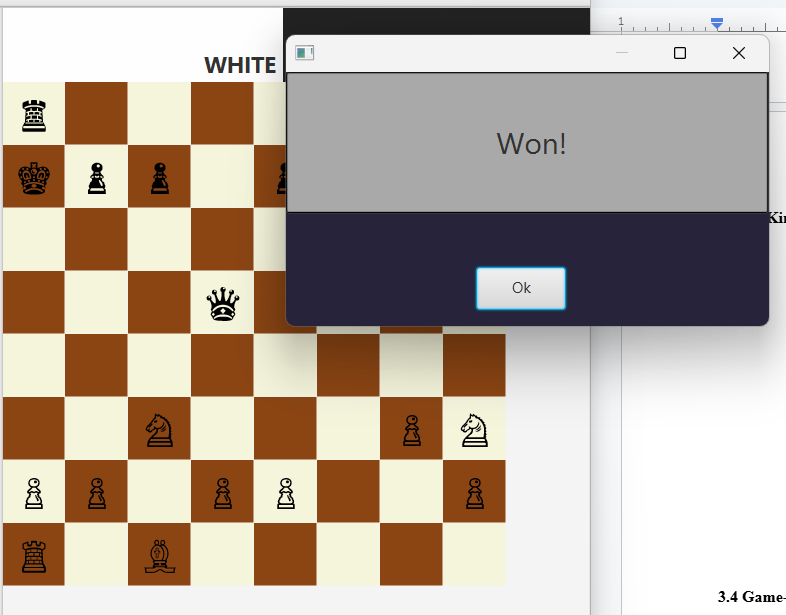


#### 3.2.6.4 King On Take

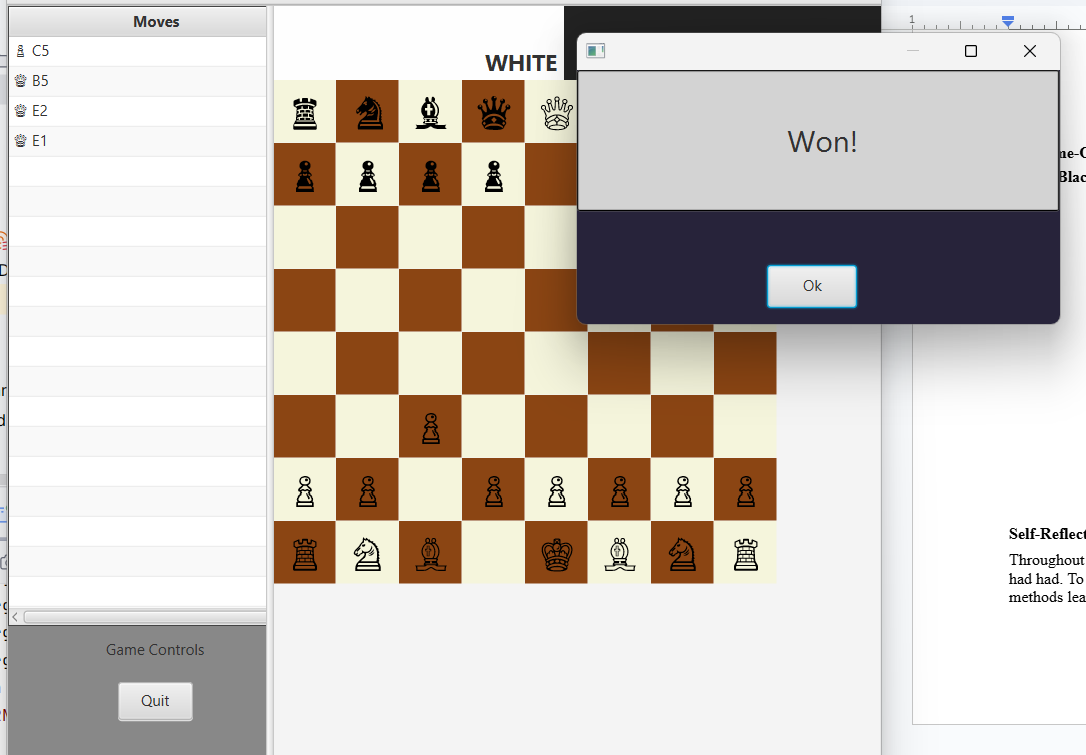


### 3.3.Game-Over Event

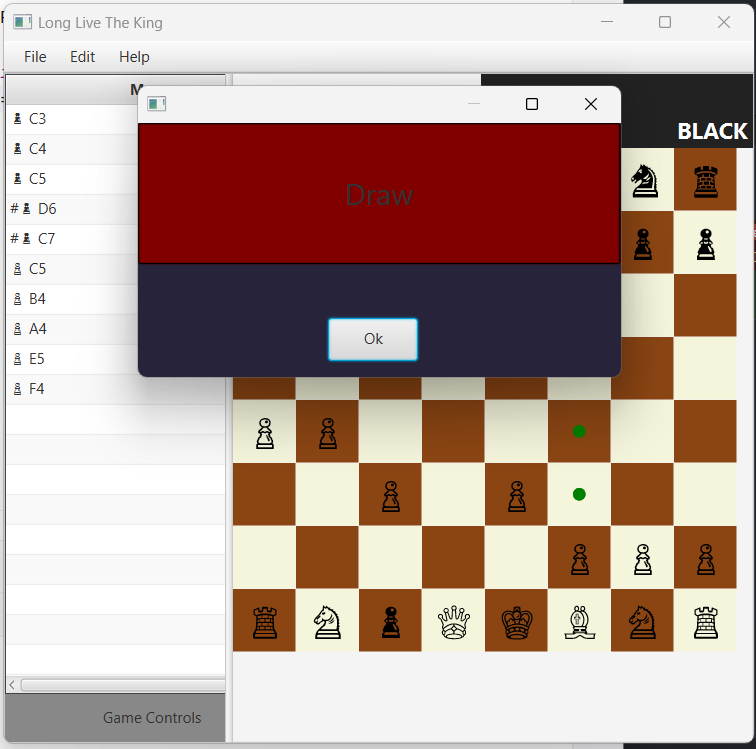
#### 3.3.4.1 Black Won

****

#### 3.3.4.2 White Won

****

#### 3.3.4.3 On Draw Event (Passive Play)

****

# 

# 4.0 Self-Reflection

Never once until this assessment, did I consider the importance of regularly modeling software systems and visualizing the system from both a top-up and bottom-up perspective, especially during development. Initially I had thought that these processes were only crucial during software requirements engineering, with UML class diagrams and flowcharts serving as mere blueprints to follow. It was until I encountered the various problems I faced in this assessment, and abandoning the code to pursue the writing the documentation and UML-class diagram that I realized the fundamental importance of these models on top of the two software-development perspectives.

At the start of this project, I had initially implemented OOP concepts following a top-down view of the system. I would first define a set of functional requirements LLtK would need to support, then a set of responsibilities LLtK would need in order to satisfy these requirements—leading to class identifications and relationship—then lower-level requirements to achieve these requirements—behaviours—and lastly variables for these lower-level requirements to operate on—attributes. Using the top-down approach, I would quickly develop a grossly-simple UML class diagram of the game, board, and chesspiece classes and chesspieces subclasses, and their respective models and controllers, and development would then feature more encapsulated and loosely-coupled classes.

However, I would learn that the top-down approach is only as good as what one knows of the system. Because I did not fully know the complexity of chess-styled games, I would miss out key and complex features, and implementing them later on would lead to these classes being tightly-coupled and equipped with ambiguous and complex relationships, and much of the project became boilerplate and hard-to-read and maintain. I then used a bottom-up view of the system, grouping up repetitions anywhere into other methods or trait-level definitions, leading to more reusable and maintainable lines of code. It was with this approach that led me to develop the generic function for child scene loading in the `*LongLiveTheKing.scala`*, and all the traits responsible for the move rules of each chess pieces*.* It was with this approach that I could implement a more typesafe, reusable, and readable project source code, implementing polymorphism, mixins, and loosely-coupling much of the project.

Even then, the bottom-up view can go so far. Trying my best to better envision the next steps, I only became unable to better visualize how to control the board model while following single responsibility principle, leading to the heavy and moderately-coupled `*BoardView.scala*` class and breach SRP for `*LongLiveTheKing.scala`*. Moreover, there still existed much of the code that still had unsafe access to other class attributes, but as submission was coming near, I could not focus on the code anymore and focused on building the final UML-class diagram and report. It was while modeling the UML class diagram, that the problems were easily exposed, offering me a better view of the project as a whole, and allowing me to easily apply the well-suited accessibility parameters, and remove redundant variables, and encapsulate a few more repeating lines of code into methods.

Nonetheless, because of the time constraints and still having more complex features to implement onto the code, the final version of LLtK is still highly limited to standard chess engines. Moreover, there still exists moderate coupling in the `*BoardView.scala*` class, and some parts of the code are not reusable. Despite the weakness of the project, using software modeling and both perspectives of software development, I was able to achieve a more readable, typesafe, less repetitive, and modular code than initially achieved, especially for chess movement logic, where both of these approaches lead to the entire scala file to be developed and tested in less than a day. Moreover, utilizing what I had learnt throughout the practicals, I was able to build a project that could handle and resist RunTimeExceptions, especially NullPointerExceptions, using the Option monad. Ultimately, this assessment made me realize the importance of constant software-modeling and looking at the system from both perspectives regularly to compile reusable snippets into methods or traits and loosely-couple most parts of the code.

# 5.0 References

*Tek Min, Object-Oriented-Programming Practicals*

*Github Copilot*