Aryaman Rawat, 2024

Final Year Project Report

A Concurrency-Based Game Environment

Aryaman Rawat

Supervisor: Luo Zhaohui

Department Of Computer Science Royal Holloway, University of London

Declaration

This report has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14509 words

Student Name: Aryaman Rawat

Date of Submission:05/04/2024

Signature: Aryaman Rawat

Table of Contents:

Abstract

A concurrency-based game environment is an interesting project that combines gaming and ever-evolving computing technology. In this plan, I will explore the exciting topic of concurrency, which allows simultaneous development of various game elements and interactions which will allow multiple games to be played at the same time in a single environment. This plan contains research information on various topics all linking to my project and eventually allowing me to successfully complete this project.

As the number of processors increases, using concurrency effectively is becoming more of a requirement. Threads will be an assured feature that will be used in my project as it will allow the development of my complex system to become more simplified, make better use of multiprocessor systems, and help make the concurrency environment more efficient and quick. Taking concurrency back to its history, in the past operating systems did not exist in computers. The computers had to execute a single program from start to end. This approach was inefficient as it didn't make optimal use of computer resources. As operating systems evolved it enabled operating systems to run multiple programs concurrently, allocating resources like memory and file handles [2].

The term thread is essential as it will be used in my project. It is a sequence of instructions given to the CPU to execute. The more threads the CPU executes the more tasks it can complete. It is essential as it can increase the speed and efficiency of multitasking [1]. Each thread has its program counter, register, and stack which keeps track of the instruction it's executing. All variables are unique to their threads and each thread cannot access other threads contents. Allowing multiple threads to run simultaneously is also called multithreading [1] [2]. This allows large-scale applications to run much faster than they would run sequentially. If executed correctly in my project it will make playing multiple games at the same time possible via concurrency.

Advantages Of Threads

Proper use of threads will reduce development and maintenance costs reducing the overall budget for any particular project. It will improve the performance of complex applications, and allow better workflow as it will help convert asynchronous workflows into sequential ones. Threads will also make codes simpler to write, read, and maintain, saving time. It will improve graphical user interface responsiveness which is ideal for my project [2].

Disadvantages Of Concurrency And Threads

With every good comes bad, threads can cause performance overhead due to multithreading which can incur overhead due to synchronization mechanisms and context switching [2]. This can lead to reduced performance for my project if it's not handled efficiently, especially in a gaming environment where real-time responsiveness is important.

Thread starvation can also occur and is a common issue in concurrent programming. This happens due to uneven thread scheduling which can lead to some threads not getting the required CPU time and thus causing thread starvation [7]. This can also cause a significant problem when the game environment is run if not handled correctly.

I will use concurrency principles to develop my application to make my final year project very efficient and quick. This will allow multiple games to be played at the same time making it a concurrent-based game environment. I will use separate threads for my graphical user interface as it's

a different process and use separate threads for the game logic for my board games for example a thread to check moves, another thread to make moves, and another to check its validity. This will allow my application to stay efficient as the graphical user interface will remain responsive even when game logic is computationally intensive. Simply meaning when multiple windows are opened for playing different games it will run its process and access its own set of data.

1 Introduction

1.1 The Problem

In the realm of game development, there are several problems when considering concurrent-based gaming environments such as tic-tac-toe and chess which introduce a set of unique challenges. Each game must operate independently and each move made in one instance shouldn't affect other instances. Similarly in a game like chess, each game needs to be run concurrently. This includes running multiple tasks at the same time such as movement for each kind of piece within the chess board, placing the pieces on the chess board, capturing the pieces and removing them from the chessboard, and timers for each player. All things mentioned above are possible through threading and multi-threading. These techniques ensure each game operates on its own thread, running concurrently yet independently.

To achieve successful implementation of threads into tic-tac-toe I need to ensure I implement threads in such a way so it's robust. One of the aims is to provide users with an easy-to-use GUI with a seamless experience for viewing and interacting. This would require a careful approach when designing the GUI since there is a change in plan from initially using JavaFX to build my GUI to using a Swing design which is a toolkit found in GUI libraries of WindowBuilder, an eclipse plugin. Since Swing GUI toolkit and WindowBuilder is a visual design tool that supports Swing, it will help me create GUI more efficiently as I can visually design the GUI within Eclipse and the tool will generate Java code for me, saving me time. Since I will be implementing my own GUI, it will be difficult to use Swing at times. To make things easier I will be using Oracle documentation for Swing and other online resources to modify my user interface further. When implementing threads with a Swing application, I need to know the kind of threads the Swing programmer deals with. Firstly initial threads are executed in the initial application code. Secondly, the (EDT) is where all the event handling code is executed, and lastly, worker threads which are also known as background threads [8]. While working with (EDT) in Swing applications I need to ensure thread safety as swing components are not thread safe meaning it should only run them on (EDT). Deadlocks are another problem to be careful of as careless use of threads can lead to dreadlocks causing the threads to be blocked and keep them waiting forever. Synchronizing is key to preventing dreadlocks.

Since I will be implementing multithreading in my Swing application, I need to be careful of some constraints to avoid problems. Time-consuming tasks shouldn't run on (EDT) to avoid unresponsiveness. Swing components should be accessed by (EDT) only [9]. Since I was initially creating tic-tac-toe which isn't a complex program, to ensure good practice I will aim to create a thread that handles lengthy tasks and another that is (EDT) to handle GUI-related activities.GUI display is another challenge since presenting information clearly to users so they understand how the game functions is important. Managing player turns, timers and buttons to ensure correct game logic is used to allow fair gameplay, and updating UI based on results introduces complexity.

The application must be able to handle multiple games being played simultaneously and synchronize between players without causing errors. Tic-tac-toe is a simple game although implementing a concurrent environment will be a challenging task. By implementing concurrency in tic-tac-toe it will create a strong helping foundation when developing a more complex board game like chess. If the above problems are handled with great care, it will ensure the user has an excellent gaming experience.

Transitioning from tic-tac-toe to developing a chess game introduces a new dimension of complexity. With the guided help of my supervisor who suggested I develop this in my second term to increase project scope, it significantly expanded the scope of my project allowing me to indulge in more complex solving problems. Unlike tic-tac-toe, chess requires managing a larger game state, including diverse piece movements, game phases (such as opening, middlegame, and endgame), and special rules (en passant, castling, promotion). My concurrent environment must efficiently handle these complexities, ensuring that simultaneous moves, game states, and player interactions are managed seamlessly across multiple instances of the game.

By incorporating the complexity and concurrency aspects identified by Coulombe[15] and Markushin[16], my approach to concurrency has been widened. This involves not just managing multiple game instances but also smartly handling the chess game's deep strategic layers and numerous possible states. I aim to leverage these insights so I can ensure the concurrent model I develop will support the demands of a complex game like chess so it can manage diverse piece movements and seamlessly synchronize game states across instances.

1.2 Aims and Goals Of The Project

Project goals

By the end of this project, I aim to have a working Java application that integrates correct concurrent methods. The user should be presented with an enjoyable and responsive GUI that will display board games to play. The game should run with correct computational and game logic. The application should run efficiently. I will ensure this is done correctly by finding helpful tutorials, and research material, or coming up with my own solutions. By integrating Java's built-in concurrency utilities, I aim to ensure that multiple game instances such as tic-tac-toe and chess, can run simultaneously without performance degradation. Since my tic-toe-toe game was a two-player game, chess will also be a two-player

For the GUI to be enjoyable and interactive, the GUI will be easy to use so it's easier for users to navigate. The GUI should allow users to click on the game mode they want to interact with. For my tic-tac-toe game, I aim to have a responsive GUI so it's more enjoyable. The user should be able to reset and exit the game at any given time. The users should also be alerted with pop-ups when the game comes to an end which is when either play wins.

After deciding the complexity of my project scope my supervisor, and I have decided to implement chess in this term to discuss my project scope and enhance overall complexity. Before the final deadline, I aim to create a working chess game that detects check and checkmate and if enough time

remains to also implement stalemate. I aim to implement most if not all chess rules according to official chess rules. I hope to implement movement logic for all the chess pieces and also implement special rules like en passant, castling, and promotion. If all of the above is achieved, my concurrent environment will allow the user to play at least two board games simultaneously in a concurrent environment. I also adhere to following correct coding standards to finish my project at the highest standard possible and come up with efficient solutions to make my project run fast and efficiently.

I am to implement most of the winning conditions from which a chess game can end. This can be a win/lose condition where the player can either checkmate, resign, or timeout. Checkmate is the most common way to end a chess game. This happens when one of the players threatens the opponent king and it cannot move to any other squares or cannot be protected by another piece and the checking piece cannot be captured. If all the conditions are met attacking player wins by checkmate. Resignation is another way a player can end the game. If users believe they are about to get checkmated soon they can choose to resign instead. Finally, timeout, timeout is also a key condition to end a game. If a player runs out of time, no matter how much advantage they've had on the board they will lose as they have run out of time [18].

For draw-based conditions due to time constraints, It won't be possible to implement stalemate, insufficient material, 50-move-rule and repetition, and agreement all at once. I aim to implement the most common method stalemate. This is when a player is left in a position where the player has no legal moves but is not in check. This is considered a stalemate and ends as a draw [18]. I will also add a button to allow users to agree to draw the game. If both players agree the game ends in a draw.

The UI for my chess game will be fixed meaning the user cannot control the size of the window frame, unlike tic-tac-toe which is responsive. This is to enhance the GUI to look more compact. I will make the UI aesthetic to ensure a seamless user experience accommodating users at all skill levels.. This will be done by using good color schemes for chess boards, pieces, and buttons. I want chess to be available for all users, beginners, intermediate, and advanced so I aim to have an instructional pop-up where users will be presented with a set of rules on how to play the games before starting the game. Since chess is more complex I will add additional rules and a special rules button that the user can open at any given time to check the rules. If users wish they should be allowed to reset the game so they can play again. I will add another button to allow the user to reset the game state when the reset button is clicked. Since most popular digital chess games have timers, I will also add timers to my game to make the game more fun and engaging. I will do this by giving the user an option to select the duration of the timer ranging from 1, 3, 5, 10, and 20 minutes. If the user wants they should also be able to create a new game from the chessboard itself. I will add a new button that lets the user create a new instance of a chess game.

To ensure I can give users a seamless experience I will make use of images in my main interface to allow users to select the game they want. To make the more interesting and engaging, I would like to add background music to the main interface and sound effects for moving pieces within the chessboard. I would also like to create a history log for players recent moves within the chess panel to allow users to see what moves they made. This will ensure users can conduct strategic analysis to allow players to review the game progression enabling them to analyze both their own and opponent's strategies. Successfully implementing this will help users identify patterns, anticipate future moves, and adjust their strategies accordingly.

Personal goals

By building the application using Java I aim to achieve a better understanding of how to develop applications and further my learning on game development using concurrent methods. This will also help me in my future professional career as I plan to become a software engineer. Getting more experience using IDEs such as Eclipse which will be used for my project will further enhance my skill set allowing me to become a better software engineer. Learning about Java mechanisms, concurrent mechanisms, threads, processes, and Swing-based applications will also help me work on a large-scale project. I also want to follow good coding standards and a high-level TDD approach using JUnit test cases to become a better software engineer.

Since expanding the scope of my project, my personal ambitions have also broadened. Developing a sophisticated game like chess requires not only an in-depth comprehension of programming principles but also an ability to navigate the intricacies of game mechanics and user interaction. This challenge will significantly improve my capabilities in game development, pushing me beyond foundational game creation toward mastering advanced concurrent methodologies. Engaging with this challenge offers a direct pathway to refine my technical skills and prepares me for a future as a proficient software engineer. Mastering concurrent methodologies which are central to modern software development will allow me to gain hands-on experience in solving real-world problems, such as ensuring responsiveness in my GUI application and managing shared resources, these are all common challenges developers face in software development and having hands-on experience will prepare me well for the future.

Working with Swing will offer me a better understanding of how I can effectively apply object-oriented design principles in creating interactive applications. Developing GUIs with swing will enhance my understanding of user interface design and event-driven programming in general which is crucial for building user-centric software. Since customer satisfaction is a top priority for any business developing interactive GUIs for my project will help users have a seamless experience and equip me with essential skills to prepare me for complex UI/UX challenges in my future career.

In addition, my personal goals are not limited to only acquiring programming skills but I also aim to be an effective problem-solver and adopt critical thinking where required to overcome challenges that I face throughout the development phases. Since this is an individual project, one of my main personal goals is to stay organized. This is for several reasons, it will help me achieve my deadlines faster and it will allow me to keep a tidy version control system which is important when working on a large-scale project with multiple developers. I aim to keep my GitLab repository clean by creating different branches for different features/functionalities. Merging the feature branch with the main one once the feature/functionality has been worked on and then deleting that feature branch from GitLab to keep a clean and tidy repository locally and remotely. Along with maintaining a clean repository, I aim to commit frequently as my code progresses and aim to write meaningful commit messages. This will help me in the future while working in big teams as I will be able to share my work with multiple team members.

I hope to improve my project planning and management skills since it's crucial for any project to succeed. This is possible through improving my documentation, debugging, and testing skills since these are vital parts of becoming a good software engineer. I want to solidify the foundational practices of becoming an accomplished software engineer. Effective documentation will not only improve my code comprehensibility but also improve my ability to communicate complex concepts. It

will help keep my project accessible and maintainable. Furthermore improving my debugging skills will help me sharpen my critical thinking and problem-solving abilities and help me dissect complex systems such as implementing a checkmate detection system. Using debugging skills I can systematically overcome challenges and ensure the reliability and user-friendliness of my application. I would also like to advance my testing skills, particularly with tools like JUnit to develop a solid approach to quality assurance. Creating regular JUnit tests will not only help me detect problems early on but will also help me understand software design principles leading to a more scalable code. Adding testing as a core part of my development process will enhance my adaptability to project changes. It will also help me facilitate agile development practices for faster and more rapid development iterations.

1.3 Survey of Related Literature

- Java Concurrency in Practice by B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea
    - This book will be used to gain a deeper understanding of concurrent programming in Java. This book introduces you to the fundamentals of threads and other advanced techniques for safely sharing data between threads and using synchronization. This book also contains various methods which will be useful in my game development. I will continue reading this book to enhance my knowledge about concurrency in Java.
- Game Programming Patterns by R. Nystrom
    - This book introduces and explains design patterns in the context of game development. It also addresses game programming challenges. The main emphasis is placed on the use of good object-oriented programming principles and how they can be applied.
- Java Design Patterns: A Tutorial by J. W. Cooper
    - This book will be heavily referenced to follow good coding procedures as it goes through some of the most common design patterns and also has examples of each type of design pattern. I will be using this book to learn more about design patterns and how I can apply them in my program.
- The Psychology of Chess and of Learning to Play It by Alfred A
    - This document introduces a beginner like me to chess in a nice manner. The document highlights chess as a complex intellectual battle and the importance of strategic planning, memory, and decision-making. Information provided within the book has played a major part in my chess game as it has helped me understand how chess works and helped me develop a chess game that is not only functionally robust but is also engaging on a psychological level.
- Rules of Chess from Wikipedia and Chess Lessons from Chess.com

    - The page from Wikipedia and insights from chess.com have significantly enhanced my understanding of the standard rules, piece movements, and special moves like castling, en passant, and pawn promotion, which are critical to replicating a realistic chess experience digitally. I have heavily used these pages to help me integrate the chess game mechanics within my application.

- Oracle Documentation on Java and Swing

    - The Oracle documentation on Swing has heavily been used for my project as it has all the resources required for building GUI for my chess application. This resource has helped me dive deep into Java's concurrency utilities, exception handling, event listeners, and more. This has enabled me to develop a robust, multi-threaded game environment with a visually appealing and intuitive GUI, incorporating features such as buttons, panels, and custom graphics for the chessboard and pieces. By using Oracle's documentation, I have been able to ensure that the application not only functions efficiently but also provides a seamless and engaging user experience which is one of my project goals.

    - The comprehensive guide has other incredible resources beyond just Swing resources, I have used it to offer me insights into efficient code structuring and optimization techniques which have been vital in writing clean, maintainable, and good-performing code, enhancing the overall quality of my application. This has allowed me to develop a solid foundation in learning Java programming principles and has offered my application a balance between aesthetic appeal and technical excellence.

- Rules of Play: Game Design Fundamentals by K. Salen and E. Zimmerman

    - The trajectory at which game-playing environments progressed is certainly astonishing. From simple text-based interfaces to complex graphically rich user interfaces showcases the impact of technological advancements on gaming. The main milestone in this evolution has been the increasing need for concurrency to manage multiple game states and real-time interactions. As outlined by K. Salen and E. Zimmerman[18], understanding the historical development underscores the present relevance of concurrency in gaming, making a rapid shift towards more immersive and interactive experiences. Diving into this book has not only broadened my perspective on game development but also raised a concern and a necessity for my project to be built in the current gaming landscape to provide users with a flawless and uninterrupted graphical user interface.

1.4 Objectives/Milestone Summary

- One of the main objectives was to design and implement an application that supports multiple concurrent game sessions without interference between them. I successfully achieved this objective by utilizing advanced concurrency control mechanisms to manage shared resources effectively ensuring data is consistent across threads.

- Another main objective was to develop a responsive and intuitive user interface that allows players to interact with the game concurrently. Whilst all my interfaces allow that to happen, I wanted to implement a more aesthetic-looking GUI. This progress was hindered due to the time constraint of the project and the complexity of the Swing framework. I did manage to allow players to interact concurrently as the feedback mechanism integer in my applications enhanced the user experience in a concurrent gaming environment.

- Additionally, I also wanted to implement game rules according to the standard and official chess and tic-tac-toe rules which I achieved. Incorporating special game rules such as castling, en passant, and pawn promotion into my chess game ensured greater user engagement.

- One of my objectives was also to follow correct testing strategies which included the TDD cycle. Writing unit tests allowed me to validate the correct game logic, concurrency controls, and UI interactions in my games.

- Finally adhering to good software engineering principles was another objective that I think I successfully managed to grasp. I aimed to document all my work from scratch, providing a clear explanation of design choices, concurrent mechanics, and thread management. Adding documentation and good coding standards allowed me to debug and fix errors early on in the stage.

2.0 Architectural paradigms and design patterns

Design Patterns

Software architecture refers to the design and organization of a software system. To make a good software architecture abstraction and decoupling are important as they help evolve the program into a better version [6]. Design patterns are one of the most important concepts in programming. A design pattern is a reliable way of reusing OO code to find solutions in a convenient way [5]. Design patterns provide better readability which helps further reduce coupling.

One of the most common frameworks, the MVC framework which consists of a controller, view, and data model tackles the concept of building a user interface. The data model consists of computational parts of the program, the view is what presents the UI and controller which helps interact between the user and the view [5]. The separations of these functions allow GUI to be accomplished. The view to model communication is essential for my application. For example, if a button is clicked an action message will be sent to the model object to get something done. To enter a new value in an entry field an update message will need to be sent to the model object to update its new value [3]. I will be using the MVC framework in my application where the model will handle all the computation parts of my project. It will hold data for the board games and perform accordingly depending on the controller's questions. The view will be responsible for the GUI and the components within my application.

The design patterns are divided into three types, creational, structural, and behavioral patterns. The creational pattern helps create objects for you instead of us instantiating objects allowing more flexibility. The structural pattern helps compose a group of objects into a larger structure and behavioral patterns help define communication between objects and the system [5].

Singleton pattern which is a type of creational pattern is essentially a class that may not have more than one instance; it provides single global point access to that instance [5]. This will be useful for my application as it will ensure the critical game components such as the resource manager have a single shared instance throughout. This will help prevent multiple buildups of resource manager objects.

A flyweight pattern which is a type of structural pattern is used for sharing objects where the instance does not contain its state but stores it externally. This allows efficient sharing of objects which saves a lot of space and is usually helpful when there exist many instances but few types [5]. This structural pattern may be used in my application to help reduce memory usage by sharing common unchanging parts of game objects across multiple instances.

Finally coming to the behavioral pattern we have an observer pattern. This pattern defines how multiple objects can be alerted of a change[5]. My application will implement this pattern as it compliments the MVC framework. It will allow game objects to receive notifications about changes in game state which is useful for handling events and updates.

3 Software Engineering

3.1 Methodology

Test-Driven-Development (TDD)

Throughout the development of my project, I have adapted a hybrid software development life cycle approach. Using a blend of extreme development with waterfalls structured planning and execution phases[19]. I have been using the TDD approach where possible where I have put the main emphasis on writing tests before writing any actual code.

The TDD cycle is a cycle of iterations referred to as 'Red-Green-Refactor' that a developer follows. The cycle consists of the following parts, firstly we start by writing a failing test, we then make the test pass by writing minimum code, once the test passes we refactor, and then we repeat the process with a new test gradually building and improving the functionality and then finally review and reflect on the test after several cycles. Using this approach has helped me create Junit test cases early on allowing me to fix errors early in the development process. Creating TDD for every functionality was not possible so I had to use a mix of extreme development as well as parallel programming to ensure I could break down large problems into smaller tasks allowing better efficiency and readability of my code. Following a test-driven development method has ensured high coding standards are followed as it has improved my code quality by keeping the code clean with fewer errors as testing has allowed me to catch errors early during my development phase. My programs have been easier to debug and maintain as I could find what's causing the error when a test fails. Since all projects will have a time constraint, using TDD has ensured faster development and has made refactoring code easier[10].

I used TDD to write my test cases for my tic-tac-toe game and all chess piece movements. For a simple game like tic-tac-toe, I used TDD in 'TicTacToeTest. Java' to code for checking initial player selection, winning conditions, draw conditions, examine buttons enablement, and disablement to check for correct functionality. These test cases ensured 'TicTacToe' game functioned as expected under various game conditions.

Additionally, for my chess pieces, I had to ensure I used TDD to allow correctness and robustness for each piece's functionality. Through TDD, tests were written before the actual implementation of the piece's logic. For each chess piece in the game, I created the corresponding test classes: "BishopTest, KingTest, KnightTest, PawnTest, QueenTest, and RookTest".These test classes were designed to validate specific movement patterns and rules associated with each piece. The test cases verified the movement of each piece for validation of legal movement within the chess board. This included testing for basic moves, capturing scenarios, and special moves.TDD also ensured error handling for my pieces is checked when an illegal move is made.TDD was also used in 'ChessPanelTest' to check for illegal king moves and to come up with check detection methods.

In summary, using TDD for developing tic-tac-toe and chess has allowed me to implement most if not all traditional rules for respective games with precision and has streamlined the development process by setting clear goals and also contributing to the overall quality and readability of each game.

Threads/Multi-Threading

In the development of TicTacToe and chess games, I have successfully managed to implement Java's concurrency mechanisms to incorporate multithreading into my games to enhance responsiveness and game logic executed smoothly in the background. This approach has been crucial in managing multiple instances of my games. Allowing the key computational aspects of the Chess game and interactive elements of Tic-Tac-Toe to run seamlessly.

Tic-Tac-Toe: Asynchronous Game Management
For my Tic-Tac-Toe game, the 'GameManager' class orchestrates the concurrent execution of multiple game instances asynchronously by utilizing 'SwingWorker' to manage game threads. This approach allowed my game to run on its own thread which was critical to prevent game execution from blocking the Swing event dispatch thread. The EDT is responsible for managing the game's GUI. By successfully handling this measure, users can launch and interact with multiple Tic-Tac-Toe games simultaneously without experiencing any issues within the user interface.

Chess: Enhanced User Interaction Through Concurrency
In my chess game, I have leveraged multithreading to separate game logic processing from the user interface, ensuring game state updates did not hinder the GUI's performance. This separation was required considering the complexity of evaluating chess moves and the need for real-time updates to the chess board.

Implementing multithreading introduced me to tough challenges. I had to ensure careful consideration was given to thread safety and synchronization of shared resources. For example, updating game states and scores required a guarantee that no two threads attempted to modify the same variables concurrently. A careful approach to this helped avoid potential inconsistencies and game errors. One of the main challenges I had to keep in mind was the management of threads and their lifecycle. I had to make sure GUI updates were performed only on EDT to maintain Swing's single-threaded model. I overcame these challenges with the use of 'SwingUtilitie.invokeLater' for chess. This allowed my GUI updates to be queued into the EDT and 'SwingWorker' for my Tic-Tac-Toe. The Oracle documentation I referred to came with a substantial amount of help allowing me to significantly improve the responsiveness and scalability of my applications. By running my game logic parallel to the GUI my project has demonstrated the effective use of concurrent programming techniques in game development to ensure a seamless and engaging user experience is given whilst handling the computational demands of processing games.

3.2 Code Snippets

This part of the section includes code snippets of the most important and interesting pieces of code within each class which I have implemented during the development phase. Due to the substantial amount of methods, I cannot add all the methods within this section. All the methods mentioned are part of the mentioned classes.

MainInterface class:

```
/**
 * Main method to launch appliciation.
 *
 * @param args the command line arguments
 */
public static void main(String[] args) {

    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                MainInterface window = new MainInterface();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}
```

The above main method is from the MainInterface class. This is the main gateway to the program. An instance of MainInterface is created. The frame visibility is set to true so the user can see the main interface.

```
 /**
 * Constructor for MainInterface class.
 */
public MainInterface() {
        gameManager = new GameManager();
        initialize();
    }
```

The constructor method above initializes a new GameManager and calls the initialize method.

```
    private void initialize() {
        // New Frame.
        frame = new JFrame("Welcome to My Game Collection");
        frame.setBounds(100, 100, 900, 600);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        JLabel welcomeLabel = new JLabel("Welcome, Please Select a Game to play!", SwingConstants.CENT
        welcomeLabel.setFont(new Font("Tw Cen MT", Font.BOLD, 24));
        frame.add(welcomeLabel, BorderLayout.NORTH);

        // Creates Jpanel.
        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(1, 2, 10, 10));
        frame.getContentPane().add(panel, BorderLayout.CENTER);

        // New button for tic-tac-toe.
        //Added images to buttons with help of oracle documentation:https://docs.oracle.com/javase%2Ft
        //Pictures taken from:https://www.cleanpng.com.
        JButton ticTacToeButton = new JButton(new ImageIcon("res/GUIImages/tic-tac-toe.png"));
        ticTacToeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                gameManager.addGame();
            }
        });
        panel.add(ticTacToeButton);

        // New button for chess game.
        JButton chessGameButton = new JButton(new ImageIcon("res/GUIImages/chessIcon.png"));
        chessGameButton.addActionListener(e -> launchChessGame());
        panel.add(chessGameButton);
        frame.setResizable(false);//Set resizable set to false as I don't want user to play on full sc


    }
```

The above private method configures the main window's properties and layout. It adds a welcome label and two buttons, each representing a game (Tic-Tac-Toe and Chess). Clicking these buttons will launch the respective game.

```
/**
 * Launches a new Chess game by creating a Che
 * This allows the Chess game to run independe
 */
public static void launchChessGame() {
    ChessGame game = new ChessGame();
    Thread gameThread = new Thread(game);
    gameThread.start();
}
```

Starts a new Chess game in a separate thread, allowing the Chess game to run independently of the main interface.

GameManager class:

```
/**
 * Adds a new TicTacToe game to the list and launches it in a separate thread.
 */
public void addGame() {
    TicTacToe newGame = new TicTacToe();
    games.add(newGame);

    //Inspired from:https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/javax/swing/SwingWorker.html.
    SwingWorker<Void, Void> gameWorker = new SwingWorker<Void, Void>() {
        @Override
        protected Void doInBackground() throws Exception {
            System.out.println("Thread ID: " + Thread.currentThread().getId()+ " is different for each game created ");
            newGame.getFrame().setVisible(true);
            return null;

        }
        @Override
        protected void done() {
            games.remove(newGame);
        }
    };

    gameWorker.execute();

}
```

The above method creates a new instance of TicTacToe which represents a new game. When the instance is created, it's added to the 'games' list which is carried out by the next line 'games.add(newGame)'.SwingWorker is used to launch each game in a separate thread. The internal method 'doInBackground()' sets the game frame to true so the user can see it. The 'done()' method removes the game from the list after it's closed.

This class is very important as it's responsible for handling multiple Tic-Tac-Toe games. The use of SwingWorker allows the game to be launched in the background which prevents freezing the main UI thread. This makes managing multiple concurrent games easier.

TicTacToe class:

```
/**
 * This class represents a thread which perfroms background tasks when TicTacToe game is being played.
 * The threads main objective is to handle player moves and determine winning player.
 * ID of current thread is printed for debugging & testing purposes.
 */
private class GameThread extends SwingWorker<Void ,Void>{

    @Override
    protected Void doInBackground() throws Exception {
        System.out.println("Thread ID: " + Thread.currentThread().getId()+ " is different for move made in each game ");
        winningPlayer();
        choosePlayer();
        return null;

    }

}
```

The above class is an internal class that extends SwingWokrer to handle game-related operations in the background. The operations being handled are game-related logic. The print message is for debugging purposes and to understand the concurrency aspects of the application.

```
/**
 * Private method which is a very important method switches between "X" and "O" accordingly for the next move.
 */
private void choosePlayer() {
    startGame = (startGame== null || startGame.equalsIgnoreCase("X")) ? "O" : "X"; // ternary conditional operator used.
}
```

This method only serves one purpose which is to alternate the current player between “X” and “O” for the next move.

```
void winningPlayer() {
    boolean isDraw = true;

    // Player X.
    if ((moves[0] == 1 && moves[1] == 1 && moves[2] == 1) ||
            (moves[3] == 1 && moves[4] == 1 && moves[5] == 1) ||
            (moves[6] == 1 && moves[7] == 1 && moves[8] == 1) ||
            (moves[0] == 1 && moves[3] == 1 && moves[6] == 1) ||
            (moves[1] == 1 && moves[4] == 1 && moves[7] == 1) ||
            (moves[2] == 1 && moves[5] == 1 && moves[8] == 1) ||
            (moves[0] == 1 && moves[4] == 1 && moves[8] == 1) ||
            (moves[2] == 1 && moves[4] == 1 && moves[6] == 1)) {
        JOptionPane.showMessageDialog(frame, "PLAYER X WINS", "Tic Tac Toe", JOptionPane.INFORMATION_MESSAGE);
        xScore1++;
        getxScore().setText(String.valueOf(xScore1));
        isDraw = false;
        disableButtons(List.of(buttons));
    }

    // Player O.
    else if ((moves[0] == 0 && moves[1] == 0 && moves[2] == 0) ||
            (moves[3] == 0 && moves[4] == 0 && moves[5] == 0) ||
            (moves[6] == 0 && moves[7] == 0 && moves[8] == 0) ||
            (moves[0] == 0 && moves[3] == 0 && moves[6] == 0) ||
            (moves[1] == 0 && moves[4] == 0 && moves[7] == 0) ||
            (moves[2] == 0 && moves[5] == 0 && moves[8] == 0) ||
            (moves[0] == 0 && moves[4] == 0 && moves[8] == 0) ||
            (moves[2] == 0 && moves[4] == 0 && moves[6] == 0)) {
        JOptionPane.showMessageDialog(frame, "PLAYER O WINS", "Tic Tac Toe", JOptionPane.INFORMATION_MESSAGE);
        oScore1++;
        getoScore().setText(String.valueOf(oScore1));
        isDraw = false;
        disableButtons(List.of(buttons));
    }

    //Checks for draw.
    if (isDraw && moves[0] != -1 && moves[1] != -1 && moves[2] != -1 &&
            moves[3] != -1 && moves[4] != -1 && moves[5] != -1 &&
            moves[6] != -1 && moves[7] != -1 && moves[8] != -1) {
        JOptionPane.showMessageDialog(frame, "IT'S A DRAW!", "Tic Tac Toe", JOptionPane.INFORMATION_MESSAGE);
    }

}
```

The above method contains game logic for tic-tac-toe. The ‘boolean isDraw’ is set to true initially assuming the game is at a draw state if any winning condition is met it’s set to false. It contains all possible winning combinations for “X” and “O”.It also handles draw scenarios. The score is updated and the winning/draw message is displayed. When the game ends buttons are disabled. The array ‘moves’ is used to represent the state of the game board. ‘1’ is used to represent an empty position whereas ‘1’ is used to represent the move made by ‘X’ and ‘0’ is the move made by ‘O’.

```
/**
 * Method to disable specified buttons.
 *
 * @param buttons the list of buttons to be disabled.
 */
void disableButtons(List<JButton> buttons) {
    for (JButton button : buttons) {
        button.setEnabled(false);
    }
}

/**
 * Method to enable specified buttons.
 *
 * @param buttons the list of buttons to be enabled.
 */
void enableButtons(List<JButton> buttons) {
    for (JButton button : buttons) {
        button.setEnabled(true);
    }
}
```

The above methods are used to disable and enable specified buttons. Setter methods are used in this case.

```
private void buttonAction(JButton button, int index) {
    if (button.getText() == null || button.getText().isEmpty()) {
        button.setText(startGame);
        if (startGame.equalsIgnoreCase("X")) {
            button.setForeground(Color.BLACK);
            moves[index] = 1;
        }
        else {
            button.setForeground(Color.BLUE);
            moves[index] = 0;
        }

        if (isFirstMove()) {
            choosePlayer();
        } else {
            GameThread gameThread = new GameThread();
            gameThread.execute();
            }

        }
    }
```

The ‘buttonAction’ method is important as it handles the action when the button is clicked. When a player makes a move the button text, color, move array, and player turns are updated. The if statement first checks if the button is empty ‘null’ and then the nested if statement is executed where the updating of button state is done. The next block of the if statement checks for the first move by calling

the ‘isFirstMove’ boolean method.If it first moves ‘choosePlayer’ method is called. This method determines which player starts the game which is done using a random assigner. The game thread is placed inside the else statement because I wanted each move to be processed in the background to handle the game logic. This also allowed Asynchronous processing since the background thread is working on game logic whilst also preventing the blocking of the main UI thread.

```
private boolean isFirstMove() {
    for (int move : moves) {
        if (move != -1) {
            return false;
        }
    }
    return true;
}
```

A private boolean method that checks if it’s the first move in the game.

```
private void initialize() {
    // Main frame.
    frame = new JFrame();
    frame.setTitle("Tic-Tac-Toe");
    frame.setBounds(100, 100, 600, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(new BorderLayout(0, 0));
    // Random assigner to get random value to first player making move.
    Random randomAssigner = new Random();
    setStartGame((randomAssigner.nextBoolean()) ? "X" : "O");

    // Creates game board panel.
    JPanel panel = new JPanel();
    panel.setBorder(new LineBorder(new Color(0, 0, 0), 2));
    frame.getContentPane().add(panel, BorderLayout.CENTER);
    panel.setLayout(new GridLayout(3, 5, 2, 2));

    createGameBoard(panel);// populates game board with buttons which user can interact.

    // Panel for player scores.
    JPanel panel_1 = new JPanel();
    panel_1.setBorder(new LineBorder(new Color(0, 0, 0), 2));
    frame.getContentPane().add(panel_1, BorderLayout.SOUTH);
    panel_1.setLayout(new GridLayout(1, 4, 2, 2));

    // Player X label and score display.
    JLabel playerX = new JLabel("Player X");
    playerX.setFont(new Font("Tw Cen MT", Font.BOLD, 30));
    playerX.setHorizontalAlignment(SwingConstants.CENTER);
    panel_1.add(playerX);

    xScore = new JTextField();
    xScore.setFont(new Font("Tw Cen MT", Font.BOLD, 40));
    xScore.setHorizontalAlignment(SwingConstants.CENTER);
    xScore.setText("0");
    panel_1.add(xScore);
    xScore.setColumns(10);

    // Player O label and score display.
    JLabel playerO = new JLabel("Player O");
    playerO.setFont(new Font("Tw Cen MT", Font.BOLD, 30));
    playerO.setHorizontalAlignment(SwingConstants.CENTER);
    panel_1.add(playerO);
```

```
oScore = new JTextField();
oScore.setFont(new Font("Tw Cen MT", Font.BOLD, 40));
oScore.setHorizontalAlignment(SwingConstants.CENTER);
oScore.setText("0");
panel_1.add(oScore);

JPanel panel_2 = new JPanel();
panel_2.setBorder(new LineBorder(new Color(0, 0, 0), 2));
frame.getContentPane().add(panel_2, BorderLayout.NORTH);
panel_2.setLayout(new BorderLayout(0, 0));

// Reset button with a action listener to reset game.
JButton btnReset = new JButton("RESET");
btnReset.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        resetGame();
    }

    private void resetGame() {
        for (JButton button : buttons) {
            button.setText("");
        }

        // Resets all buttons.
        for (int i = 0; i < moves.length; i++) {
            moves[i] = -1;
        }

        enableButtons(List.of(buttons));
    }
});
btnReset.setBackground(new Color(255, 255, 255));
btnReset.setFont(new Font("Tw Cen MT", Font.BOLD, 30));
panel_2.add(btnReset, BorderLayout.CENTER);

JPanel panel_3 = new JPanel();
panel_3.setBorder(new LineBorder(new Color(0, 0, 0), 2));
frame.getContentPane().add(panel_3, BorderLayout.EAST);
panel_3.setLayout(new BorderLayout(0, 0));
```

```
//Code inspiration link: https://www.tutorialspoint.com/swingexamples/show_confirm_dialog_with_yesno.htm.
JButton btnExit = new JButton("EXIT");
btnExit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        exitGame();
    }

    private void exitGame() {
        int user_option = JOptionPane.showConfirmDialog(frame, "Are you sure you want to exit?", "Tic Tac Toe", JOptionPane.YES_NO_OPTION);
        if (user_option == JOptionPane.YES_OPTION) {
            frame.dispose();
        }

    }
});
btnExit.setBackground(new Color(255, 255, 255));
btnExit.setFont(new Font("Tw Cen MT", Font.BOLD, 30));
panel_3.add(btnExit, BorderLayout.CENTER);
xScore.setEditable(false); //Editable property set to false so user cannot modify it.
oScore.setEditable(false);
```

```
private void createGameBoard(JPanel panel) {
    for (int i = 0; i < buttons.length; i++) {
        buttons[i] = new JButton("");
        int creation = i;
        buttons[i].addActionListener(e -> buttonAction(buttons[creation], creation));
        buttons[i].setFont(new Font("Tw Cen MT", Font.BOLD, 80));
        buttons[i].setBackground(new Color(255, 255, 255));
        panel.add(buttons[i]);
    }
}
```

The above snippets are all part of the 'initialize' method inside the TicTacToe class. This method is accountable for setting up the main frame, creating panels, labels, and buttons, and initializing the tic-toe-toe game board. Within the method, there are other private methods such as 'resetGame' which resets the game by clearing up all the button texts and enabling the buttons to allow the user to play again. The 'exit game ()' contains the variable 'user_option' which displays a confirmation dialog before the user exits the game. Finally, the 'createGameBoard' method uses for loop to populate the game board with buttons. It iterates through each element in the 'buttons' array. Each element represents a button in the game. After each button is configured, it is added to the panels.

ChessBoard class:

This class represents the game board of my chess application. It initializes the chess pieces in standard starting positions and manages the rendering of the board and its pieces.

```
/**
 * This method draws a 8x8 chess board.
 * This class was also modified using documentation provided by oracle at:https://docs.oracle.com/ja
 * @param g The Graphics object used for drawing.
 */
private void drawChessBoard(Graphics g) {
    //Paint method uses nested for loop to iterate through each square on chess board.
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            // Chess board inspired from the video link: https://www.youtube.com/watch?v=vO7wHV0HB8w
            if ((x + y) % 2 == 0) {
                g.setColor(new Color(235, 236, 208));
            } else {
                g.setColor(new Color(119, 154, 88));
            }               g.fillRect(x * SQUARE_SIZE, y * SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE);

        }

    }


}
```

```
private void initializeWhitePieces() {
    // White pieces.
    chessPieces.add(new Rook(this, 0, 7, ChessPanel.WHITE));
    chessPieces.add(new Rook(this,7, 7, ChessPanel.WHITE));
    chessPieces.add(new Bishop(this,2, 7, ChessPanel.WHITE));
    chessPieces.add(new Bishop(this,5, 7, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,0, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,1, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,2, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,3, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,4, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,5, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,6, 6, ChessPanel.WHITE));
    chessPieces.add(new Pawn(this,7, 6, ChessPanel.WHITE));
    chessPieces.add(new King(this,4, 7, ChessPanel.WHITE));
    chessPieces.add(new Queen(this,3, 7, ChessPanel.WHITE));
    chessPieces.add(new Knight(this,1, 7, ChessPanel.WHITE));
    chessPieces.add(new Knight(this,6, 7, ChessPanel.WHITE));
}
```

The above image where the drawChessBoard method is shown is an important piece of this class as it draws an 8x8 chessboard by altering the colors of the squares. Furthermore, the image below shows how pieces are initialized within the class. An array list of chess pieces is used which contains all the chess pieces. Initialization methods set up white and black pieces on the board and place them in the exact position used in traditional chess.

```
/**
 * Resets chessboard to orignal state.
 */
public void resetChessBoard() {
    //Clear the existing pieces
    chessPieces.clear();
    //Reinitialize white pieces
    initializeWhitePieces();
    //Reinitialize black pieces
    initializeBlackPieces();
}
```

This method is important to reset the current state of the board to its original state when the user wants to start a new game or reset the game.

Type class:

```
public enum Type {
    KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN //Th
}
```

This class defines the types of chess pieces available in the game. Each enum value represents the type of piece within the chess board consisting of: “KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN”. These are the chess pieces represented within the game, used to differentiate piece types and implement specific movements and capturing rules for each type.

ChessGame class:

```
*/
public class ChessGame implements Runnable {
    /**
     * Main method that serves as the entry point of the applicatior
     * It schedules the application to be run using the Event Dispat
     * to ensure thread safety with Swing components.
     *
     * @param args not used.
     */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new ChessGame());//  // Main clas

    }

    /**
     * Sets up and displays the main game window for the chess game.
     * It initialises the frame, sets its properties, adds a ChessPa
     * and makes the frame visible to the user. This method is inter
     */
    public void run() {
        JFrame frame = new JFrame("Chess");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setLayout(new BorderLayout());// Adding new layout mar

        ChessPanel pn = new ChessPanel();
        frame.add(pn);// Add the ChessPanel to the frame

        frame.pack();// Resize frame to fit the preferred sizes of i
        frame.setVisible(true);// Set the frame to be visible
        frame.setMinimumSize(frame.getPreferredSize());// Set a mini

        frame.setResizable(false);//Set resizable set to false as I

        pn.runGame();
    }

    /**
     * Utility method to launch a new game instance.
     * It schedules a new ChessGame instance to be started on the ED
     */
    public static void launchNewGame() {
        SwingUtilities.invokeLater(new ChessGame());
    }
}
```

The above image represents the main entry point for the chess game. It displays the game window initiates the game component and starts the game loop.

- The Main method is similar to MainInterface, it uses SwingUtilities.invokeLater to ensure that the Chess game's GUI is created and updated on the EDT.
- The run method sets up the main game window for Chess, including initializing the JFrame, setting its properties such as size, visibility, and layout, and adding a ChessPanel to it for rendering the game.
- Finally, the launchNewGame method offers a utility to launch a new Chess game instance on the EDT, allowing for multiple games to be started independently.

MouseController class:

The purpose of this class is to control mouse interaction within the ChessPanel, this includes piece selection and movement. It extends 'MouseAdapter' which is a key class for receiving mouse events.

```
@Override
public void mouseDragged(MouseEvent e) {
    x = e.getX();
    y = e.getY();
}
```

```
@Override
public void mousePressed(MouseEvent e) {
    pressed = true;
}
```

```
@Override
public void mouseReleased(MouseEvent e) {
    pressed = false;
    Piece chosenPiece = chessPanel.getChosenPiece();
    if (chosenPiece != null) {
        chosenPiece.x = (x / ChessBoard.SQUARE_SIZE) * ChessBoard.SQUARE_SIZE;
        chosenPiece.y = (y / ChessBoard.SQUARE_SIZE) * ChessBoard.SQUARE_SIZE;
        chosenPiece = null; // Reset chosenPiece
    }
}
```

The above mouse event handlers are very important to my chess game. The 'mouseDragged' handler updates the position variables for the mouse when the mouse is dragged. This is essential for tracking piece movement. The 'mousePressed' handler sets the boolean flag 'pressed' to true indicating piece selection or movement initiation. Finally, the 'mouseReleased' handler resets the flag back to false this ensures the piece movement logic as it puts the piece to the nearest square.

ChessPanel class:

This class serves as the core panel for the chess game. It extends the 'JPanel' and implements 'Runnable' for managing the game loop. This class handles piece movement, game rendering, game logic, and user interaction through mouse events.

```
// Game state variables
int presentColor = WHITE; // Present
private Piece chosenPiece; // Curren
public static Piece castlingPiece; /
private Piece checkingPiece;// Piece
boolean isStalemate; //Boolean flag
boolean gameFinished; //Boolean flag
boolean validMove;// boolean variabl
boolean validSquare;// boolean varia
```

The variables shown above are vital for this class as they track the current game state including active player ‘white’ or ‘black’, whether the game has finished if a valid move or valid square is chosen, and if a stalemate or checkmate has occurred.

Important Rule Validation and Chess Mechanics methods within the ChessPanel class:

```
boolean isKingInCheck(boolean opponent) {
    Piece king = getKing(opponent); // false to get the current player's king
    if (king == null) {
        return false;
    }
    for (Piece piece : board.getChessPieces()) {
        if (piece.color != presentColor && piece.canMove(king.column, king.row)) {
            checkingPiece = piece;
            return true;
        }
    }
    checkingPiece = null;
    return false;
}
```

The isKingInCheck method checks if the king of either of the players in turn to the opponent based on the opponent's flag is in a state of ‘check’.

```
boolean willKingBeInCheck(Piece king) {
    if (king == null) {
        return false;
    }
    for (Piece piece : board.getChessPieces()) {
        if (piece.color != presentColor && piece.canMove(king.column, king.row)) {
            checkingPiece = piece;
            //System.out.println("King"+king+" is in check by " + piece);
            return true;
        }
    }
    checkingPiece = null;
    return false;
}

//This method finds and returns the king piece for the current player or their oppo
/**
 * @param opponentPiece
 * @return
 */
private Piece getKing(boolean opponentPiece) {
    Piece king = null;//Inital king peice set to null.
    for(Piece piece: board.chessPieces) {// For loop used to iterate over all piece
        if(opponentPiece) {//If true it searches for opponent's king piece.
            if(piece.type == Type.KING && piece.color != presentColor) {
                king = piece;
            }
        }
        else{// Looks for current players king piece if flag is false.
            if(piece.type == Type.KING && piece.color == presentColor) {
                king = piece;//returns found king piece.
            }

        }

    }
    return king;//Returns null if no king is found.
}
```

The above two methods are crucial for the chess game as it is required for checking for checkmate conditions. The willKingBeInCheck method will determine if a move would place King in check. The getKing method acts as a getter method to retrieve the King piece for the current player or their opponent.

```
 */
public boolean illegalKingMove(Piece king){
    if (king == null) {
        return false;
    }
    if(king.type == Type.KING){
        for(Piece piece : board.chessPieces){
            if(piece.canMove(king.column,  king.row) && piece != king && piece.color != king.color) {
                return true;
            }
        }
    }
    return false;
}
```

The illegalKingMove method is another vital method that is required to verify if the move is illegal due to placing or leaving the king in check.

```
public boolean hasSafeMoves(Piece king) {
    //System.out.println("Checking safe moves for King at: " + king.column + ", " +
    for (int columnOffset = -1; columnOffset <= 1; columnOffset++) { // Iterate over
        for (int rowOffset = -1; rowOffset <= 1; rowOffset++) { // Iterate over all
            int newColumn = king.column + columnOffset;
            int newRow = king.row + rowOffset;
            // Ensure new position is within the board.
            if (newColumn >= 0 && newColumn < 8 && newRow >= 0 && newRow < 8) {
                // Clone king to simulate the move.
                Piece temp = king.clone();
                temp.column = newColumn;
                temp.row = newRow;
                //System.out.println("Testing move to: " + newColumn + ", " + newRow
                if (temp.canMove(newColumn, newRow)) {
                    //System.out.println("Move is within legal movement rules.");
                    if (!willKingBeInCheck(temp)) {
                        //System.out.println("This move is safe from checks.");
                        return true; // King has at least one safe move.
                    } else {
                        //System.out.println("This move would place the King in chec
                    }
                } else {
                    //System.out.println("Move is not within legal movement rules.")
                }
            }
        }
    }
    //System.out.println("No safe moves available for the King.");
    return false; // No safe moves available.
```

This method is one of the most complex methods and one of the crucial methods required to work perfectly to achieve checkmate. This method checks for the existence of any legal moves that the king can make without placing itself into check.
Here's how it works:

- Iterate through potential moves by looping over row and column offsets from the king's current position.
- For each potential move, the method simulates the move by creating a temporary clone of the king in a new position this is a crucial stage as it doesn't affect the actual game state.
- It then carries a safety check for each simulated move, the method checks if moving the king to this new position would place it under attack by any of the opponent's pieces. This is carried out by evaluating if any of the opponent pieces can legally move to the king's new position.
- Finally, the method determines the legality and safety of the piece. If a legal move exists it doesn't place the king in check, the method finalizes that the king has safe moves and returns true. If no such moves are found it returns false indicating the king has no safe moves.

As shown in this image on the left the King piece is in check and has picked an illegal square therefore user is not allowed to place it there. In a similar scenario, the king piece can legally move to the square when in check shown in the image to the left.

```
public boolean canBlockOrCaptureCheckingPiece(Piece king) {
    // Iterate over all chess pieces on the board.
    for (int index = 0; index < board.getChessPieces().size(); index++) {
        Piece piece = board.getChessPieces().get(index);
        // Check if the piece is an ally of the king
        if (piece.color == king.color) {
            // Iterate over all squares on the chess board to explore possible moves
            for (int column = 0; column < 8; column++) {
                for (int row = 0; row < 8; row++) {
                    // Check if the ally piece can legally move to the current square
                    if (piece.canMove(column, row)) {
                        // If the square is occupied by the checking piece, capturing it w
                        if (checkingPiece.row == row && checkingPiece.column == column) {
                            // Debug print statement for capturing the checking piece.
                            //System.out.println(piece.type + " Can capture checking piece
                            return true;
                        }
                        // Clone the piece to simulate the move without affecting the game
                        Piece temp = piece.clone(); // Clone method is used to avoid mutat
                        temp.column = column;
                        temp.row = row;
                        // Temporarily move the piece in the simulation.
                        board.getChessPieces().set(index, temp);
                        // Check if moving the piece resolves the check against the king.
                        if (!isKingInCheck(false)) {
                            // Debug print statement for a move that blocks the check or c
                            //System.out.println(temp.type + " Can block or capture checki
                            // Restore the original piece after the simulation.
                            board.getChessPieces().set(index, piece);
                            return true; // The check can be resolved by this piece's move
                        }
                        // Restore the original piece if the simulated move does not resol
                        board.getChessPieces().set(index, piece);
                    }
                }
            }
        }
    }
    return false; // Return false if no piece can block the check or capture the checking
}
```

This method in the image above is also important as it evaluates whether any piece other than the king can move in a way to either capture or block the checking piece. In the game of chess, it is vital for a check to be countered without resorting to moving the king itself.
Here is how canBlockOrCaptureCheckingPiece works:

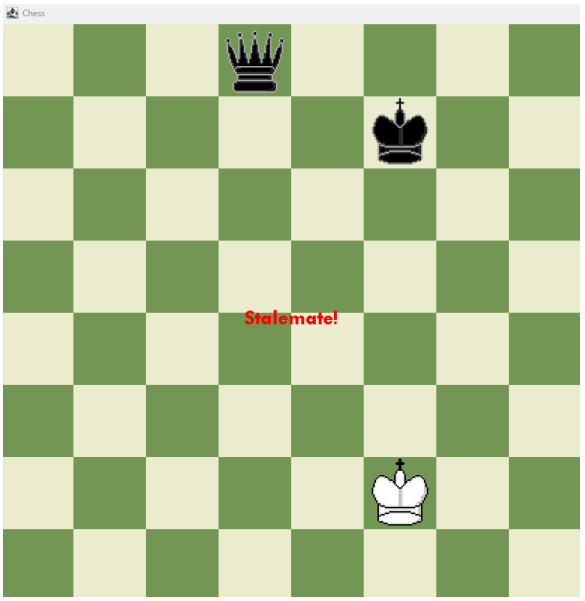
- First, it identifies the checking piece that has put the king in check.
- It then evaluates ally moves by iterating over all pieces on the player's side except the king.
- For each ally piece, it then checks every possible move to see if it can capture the checking piece or move to a square that blocks the check.
- Similar to the previously mentioned 'hasSafeMoves' method, this one involves simulation without altering the actual game state.
- Finally, if the ally piece can capture the checking piece or block its attack thereby resolving check method returns true otherwise it returns false.

```
*/
public boolean isCheckmate() {
    Piece king = getKing(false);
    if(isKingInCheck(false)) {// If king is not in check and if ally piece
        if(!canBlockOrCaptureCheckingPiece(king) && !hasSafeMoves(king)) {
            //System.out.println("Checkmate!");
            return true;
        }
    }
    return false;//Not a checkmate
}
```

```
private boolean isStalemate() {
    int pieceCount = 0; // Initialize a counter for
    // Log the start of a stalemate check for debugg
    //System.out.println("Checking for stalemate for
    for(Piece piece : board.chessPieces) { // Counts

        if(piece.color == presentColor) {
            pieceCount++;
        }
    }
    //System.out.println("Piece count: "+pieceCount)

    // Keeps count of number of pieces current playe
    if(pieceCount == 1) { // If the current player h

        if(!hasSafeMoves(getKing(false))){ // Retrie

            isStalemate = true;
            return true;
        }
    }
    return false;
}
```

Both 'isCheckmate' and 'isStalemate' methods are crucial for determining the endgame conditions in chess. The 'isCheckmate' method determines whether the game has reached a checkmate condition. Checkmate occurs when the king is in a position to be captured and cannot escape the threat of

capture on the next move. There's no legal move that the player can make to remove their king from threat. The isStalemate method checks for a stalemate condition. Stalemate is a draw-based situation where the player whose turn it is cannot make any legal move and their king is not in check. It's a draw condition, indicating the game ends without a winner. Below the images show the state of the game after the user wins by checkmate and when it’s a stalemate.



Piece class:

```
@Override
public Piece clone(){
    Piece piece = new Piece(board, column, row, color);
    piece.x = x;
    piece.y = y;
    piece.previousColumn = previousColumn;
    piece.previousRow = previousRow;
    piece.hasPieceMoved = hasPieceMoved;
    piece.enPassantEligible = enPassantEligible;
    piece.board = board;
    piece.capturedPiece = capturedPiece;
    piece.image = image;
    piece.type = type;
    return piece;// Returns new cloned Piece object.
}
```

The clone method above has an important purpose in allowing the simulation to occur without altering the game's actual state. This is useful for checking the outcomes of moves' legality and for the

outcomes of moves without affecting ongoing games. The method returns a new ‘Piece’ object with identical properties to the original.

```
public BufferedImage loadImage(String imageLoader) {
    BufferedImage image = null;

    try {
        image = ImageIO.read(getClass().getResourceAsStream(imageLoader + ".png"));
    }catch(IOException e) {
        e.printStackTrace();
    }
    return image;

}
```

The loadImage method is another important method to load images from the resource folder. This method helps visually represent the chess piece on the board. The method uses ImageIO.read to load a BufferedImage from a given path, handling IOException if the file cannot be found or read. The method returns the loaded image for rendering.

```
/**
 * Updates piece position aftwr user has made a mov
 */
public void UpdatePiecePosition() {
    if(type == Type.PAWN) {
        if(Math.abs(row - previousRow) == 2) {
            enPassantEligible = true;
        }
    }
    x = getX(column);
    y = getY(row);
    previousColumn = getColumn(x);
    previousRow = getRow(y);
    hasPieceMoved = true;

}
```

This method updates the piece's position after a move has been made, handling special cases like pawn's first move for en passant eligibility. It updates the pieces pixel and board coordinates to the new position and sets hasPieceMoved to true. For pawns moving two squares, it also sets enPassantEligible to true.

```
public boolean bishopDiagonalCheck(int targetColumn, int targetRow) {
    // Checks for down right movement
    for(int r = previousRow + 1, col = previousColumn + 1; r < targetRow && col < targetColumn; r++, col++) {
        for(Piece piece : board.getChessPieces()) {
            if(piece.column == col && piece.row == r) {
                capturedPiece = piece;
                return true;
            }
        }
    }
    // Checks for down left movement
    for(int r = previousRow + 1, col = previousColumn - 1; r < targetRow && col > targetColumn; r++, col--) {
        for(Piece piece : board.getChessPieces()) {
            if(piece.column == col && piece.row == r) {
                capturedPiece = piece;
                return true;
            }
        }
    }
    // Checks for up left movement
    for(int r = previousRow - 1, col = previousColumn - 1; r > targetRow && col > targetColumn; r--, col--) {
        for(Piece piece : board.getChessPieces()) {
            if(piece.column == col && piece.row == r) {
                capturedPiece = piece;
                return true;
            }
        }
    }
    // Checks for up right movement
    for(int r = previousRow - 1, col = previousColumn + 1; r > targetRow && col < targetColumn; r--, col++) {
        for(Piece piece : board.getChessPieces()) {
            if(piece.column == col && piece.row == r) {
                capturedPiece = piece;
                return true;
            }
        }
    }



    return false;
}
```

One of the most interesting methods which I had to implement was for the bishop piece. This method checks for clear diagonal paths for bishops. The method ensures no pieces block the path to the target position. It’s similar to another method isPathClear but checks diagonally. Iterates through diagonal squares between the bishop's current and target positions.

Bishop class:

The ‘Bishop’ class in my chess game is a subclass of the ‘Piece’ class. Every other piece is also a subclass of the ‘Piece’ class and due to this section getting very lengthy, I will not be adding snippets to other Piece classes. The purpose of this class is to create a new instance of a ‘Bishop’ piece with a specified starting position and color. It calls the superclass Piece constructor to set the initial position and color, sets the piece's type to BISHOP, and loads the appropriate image based on the color.

```
public Bishop(ChessBoard board, int column, int row, int color) {
    super(board,column, row, color);// Superclass constructor to set position and color.
    type = Type.BISHOP; //Setting type for this piece.

    //Loads image for Bishop piece.
    if (color == ChessPanel.WHITE) {
        image = loadImage("/pieces/w-bishop");
    }
    else {
        image = loadImage("/pieces/b-bishop");
    }


}

/**
 * This boolean method determines if bishop piece can move to it's target location.
 * The piece can move if it moves diagonally, the path is unobstructed,
 * and follows the general chess movement rules.
 * @param targetColumn The target column position for Bishop's move.
 * @param targetRow The target row position for Bishop's move.
 * @return true if move is valid otherwise return false.
 */
public boolean canMove(int targetColumn, int targetRow) {
    if(isInsideBoard(targetColumn, targetRow) && isPieceInSameSquare(targetColumn, targetRow)== false) {
        if(Math.abs(targetColumn - previousColumn) == Math.abs(targetRow - previousRow)) {
            if(canCapture(targetColumn, targetRow) && bishopDiagonalCheck(targetColumn, targetRow)== false) {
                return true;
            }

        }
    }
    return false;
}
```

This Bishop class is structured similarly to other chess piece classes in my project. Each piece class (like Rook, Knight, Queen, etc.) follows the same basic structure but implements its unique canMove logic according to the rules of chess for that specific piece.

3.3 Testing

This section contains test codes that I had to create over time for my tic-tac-toe game and chess game, implementing different features. It is essential to ensure testing is carried out to achieve most of the required functionalities and to make it work as intended. Due to numerous test cases and test files, it will be unrequired to add all the test cases and test files as all pieces have been developed using TDD but all use different logic to move and capture.I will provide some examples of test files and cases for my chess development to show how I carried TDD to implement these methods successfully.

```
@Test
public void testChoosePlayer() {
    // Test to see that the first move is random
    String player = ticTacToe.getStartGame();
    assertEquals("X".equals(player) || "O".equals(player), true);
}
```

In this test case, it checks for random “X” and “O” assignments when the user makes the first move. The ‘assertEquals’ statement checks if the starting player is either “X” or “O”.

```
@Test
public void testHorizontalX() {
    // Test to see if player X wins in the first row and pop up message shows.
    ticTacToe.getMoves()[0] = 1;
    ticTacToe.getMoves()[1] = 1;
    ticTacToe.getMoves()[2] = 1;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getxScore().getText());
}

@Test
public void testHorizontalX2() {
    // Test to see if player X wins in the second row and pop up message shows.
    ticTacToe.getMoves()[3] = 1;
    ticTacToe.getMoves()[4] = 1;
    ticTacToe.getMoves()[5] = 1;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getxScore().getText());
}

@Test
public void testHorizontalX3() {
    // Test to see if player X wins in the third row and pop up message shows.
    ticTacToe.getMoves()[6] = 1;
    ticTacToe.getMoves()[7] = 1;
    ticTacToe.getMoves()[8] = 1;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getxScore().getText());
}
```

This test case tests if Player X wins in the first, second, and third horizontal rows. The getter method is used to get the moves used for testing. Assertion checks if player ‘X’ score is updated after the moves have been made. Similar test cases are made to test vertical and diagonal winning combinations for player ‘X’.All the possible winning combinations logic is stored in the

'winningPlayer' method. It contains winning combinations for 'X' and 'O' and also checks possible combinations that could also result in a draw. The winning combinations are checked using the value 'moves' array at the specified positions and if a winning combination is found for either 'X' or 'O', that player wins or if no winning combination is found, it ends in a draw.

```
@Test
public void testHorizontalO() {
    ticTacToe.getMoves()[0] = 0;
    ticTacToe.getMoves()[1] = 0;
    ticTacToe.getMoves()[2] = 0;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getoScore().getText());
}

@Test
public void testHorizontalO2() {
    ticTacToe.getMoves()[3] = 0;
    ticTacToe.getMoves()[4] = 0;
    ticTacToe.getMoves()[5] = 0;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getoScore().getText());
}

@Test
public void testHorizontalO3() {
    ticTacToe.getMoves()[6] = 0;
    ticTacToe.getMoves()[7] = 0;
    ticTacToe.getMoves()[8] = 0;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getoScore().getText());
}

@Test
public void testVerticalO() {
    ticTacToe.getMoves()[0] = 0;
    ticTacToe.getMoves()[3] = 0;
    ticTacToe.getMoves()[6] = 0;
    ticTacToe.winningPlayer();
    assertEquals("1", ticTacToe.getoScore().getText());
}
```

The test cases above, test for Player 'O' winning combinations. Everything is similar to player 'X' test cases the only difference is instead of getxScore(), it's getoScore() to check if the player 'O' score is updated, and the value in the statement ticTacToe.getMoves()[] = 1; changes to 0. Again similar tests for Player 'O' are carried out to check vertical and diagonal combinations.

```java
@Test
public void testGameDraw() {
    TicTacToe game = new TicTacToe();

    // Results in a draw
    game.getMoves()[0] = 1;
    game.getMoves()[1] = 0;
    game.getMoves()[2] = 1;
    game.getMoves()[3] = 0;
    game.getMoves()[4] = 1;
    game.getMoves()[5] = 0;
    game.getMoves()[6] = 0;
    game.getMoves()[7] = 1;
    game.getMoves()[8] = 0;

    game.winningPlayer();

    // Score should remain 0 if the game has resulted in a draw.
    assertEquals(0, Integer.parseInt(game.getxScore().getText()));
    assertEquals(0, Integer.parseInt(game.getoScore().getText()));
}
```

The testGameDraw() test case checks if the game correctly detects a draw. This is ensured by checking both player ‘X’ and player ‘O’ score and their scores should remain 0.

```java
@Test
public void testEnableButtons() {
    // Creating buttons 1-9 for testing purposes
    JButton btn1 = new JButton();
    JButton btn2 = new JButton();
    JButton btn3 = new JButton();
    JButton btn4 = new JButton();
    JButton btn5 = new JButton();
    JButton btn6 = new JButton();
    JButton btn7 = new JButton();
    JButton btn8 = new JButton();
    JButton btn9 = new JButton();
    List<JButton> buttons = Arrays.asList(btn1, btn2, btn3, btn4, btn5, btn6, btn7, btn8, btn9);
    ticTacToe.disableButtons(buttons); // Disables the buttons first
    ticTacToe.enableButtons(buttons); // Enables the buttons

    // Checks if buttons are enabled
    assertTrue(btn1.isEnabled());
    assertTrue(btn2.isEnabled());
    assertTrue(btn3.isEnabled());
    assertTrue(btn4.isEnabled());
    assertTrue(btn5.isEnabled());
    assertTrue(btn6.isEnabled());
    assertTrue(btn7.isEnabled());
    assertTrue(btn8.isEnabled());
    assertTrue(btn9.isEnabled());
}
```

Tests for ‘enableButtons’ method in the TicTacToe class. It should enable the list of buttons given. Adding assertion checks if all the buttons in the provided list are enabled after calling the method. A similar test is carried out for ‘disableButtons’ but instead it calls the ‘disableButtons’ method to check if all buttons have been disabled.

```java
package aryaman_Project;

import static org.junit.Assert.*;

public class ChessPanelTest {

    @Test
    public void testIllegalKingMove() {
        // Setup the chess panel and clear any existing pieces.

        ChessPanel chessPanel = new ChessPanel();
        chessPanel.board.getChessPieces().clear();
        // Place a black king and a white rook on the board such that the rook checks the king.
        King kingBlack = new King(chessPanel.board, 4, 0, ChessPanel.BLACK);
        Rook rookWhite = new Rook(chessPanel.board, 4, 1, ChessPanel.WHITE);
        chessPanel.board.getChessPieces().add(kingBlack);
        chessPanel.board.getChessPieces().add(rookWhite);
        // Assert that the move is correctly identified as illegal king would be in check.
        boolean result = chessPanel.illegalKingMove(kingBlack);
        assertTrue("The move should be identified as illegal as it puts king in check", result);
    }

    @Test
    public void testGameDetectsCheck() {
        // Setup the chess panel and clear any existing pieces
        ChessPanel chessPanel = new ChessPanel();
        chessPanel.board.getChessPieces().clear();

        // Place a white king and a black rook on the board in positions that result in a check
        King kingWhite = new King(chessPanel.board, 4, 0, ChessPanel.WHITE);
        Rook rookBlack = new Rook(chessPanel.board, 4, 7, ChessPanel.BLACK);
        chessPanel.board.getChessPieces().add(kingWhite);
        chessPanel.board.getChessPieces().add(rookBlack);

        // Assert that the game correctly detects the king is in check
        assertTrue("Game should detect check on the white king", chessPanel.isKingInCheck(false));
    }
}
```

This test file is a crucial part of ensuring the reliability of the chess game's logic. By following TDD, I ensured strong verification of game rules regarding illegal moves and check detection helped me maintain game integrity and helped me quickly identify errors introduced during development. Documenting such tests adds value by providing clear examples of how the game logic should handle specific scenarios, facilitating easier maintenance and understanding of the code.
The first test case creates a scenario where the move made king piece should result in an illegal move. For that result, assertTrue is used. The next test case checks if a king is in check or not. Pieces have been placed in a way that should place the king in check.

```java
public class PawnTest {

    //White pawn testing
    @Test
    public void testPawnMovementOneSquareWhite() {
        ChessBoard board = new ChessBoard();
        Pawn pawn = new Pawn(board, 1, 4, ChessPanel.WHITE);
        assertTrue("Pawn should move forward one square", pawn.canMove(1, 3));
    }

    @Test
    public void testPawnInitialMovementWhite() {
        ChessBoard board = new ChessBoard();
        Pawn pawn = new Pawn(board, 1, 6, ChessPanel.WHITE); // Starting from row 6 for white
        assertTrue("Pawn should move forward two squares if it's the first move", pawn.canMove(1, 4));
    }

    @Test
    public void testPawnInvalidMoveWhite() {
        ChessBoard board = new ChessBoard();
        Pawn pawn = new Pawn(board, 1, 6, ChessPanel.WHITE);
        assertFalse("Pawn should not move backwards", pawn.canMove(1, 7));
    }

    @Test
    public void testPawnCapturesDiagonallyWhite() {
        ChessBoard board = new ChessBoard();
        Pawn pawn = new Pawn(board, 1, 6, ChessPanel.WHITE);
        Pawn enemyPawn = new Pawn(board, 2, 5, ChessPanel.BLACK);
        board.getChessPieces().add(pawn);
        board.getChessPieces().add(enemyPawn);
        assertTrue("Pawn should be able to capture diagonally", pawn.canMove(2, 5));
    }

    @Test
    public void testPawnMoveIfBlockedWhite() {
        ChessBoard board = new ChessBoard();
        Pawn pawn = new Pawn(board, 1, 6, ChessPanel.WHITE);
        Pawn blockingPawn = new Pawn(board, 1, 5, ChessPanel.WHITE);
        board.getChessPieces().add(pawn);
        board.getChessPieces().add(blockingPawn);
        assertFalse("Pawn should not be able to move forward if blocked", pawn.canMove(1, 5));
    }
```

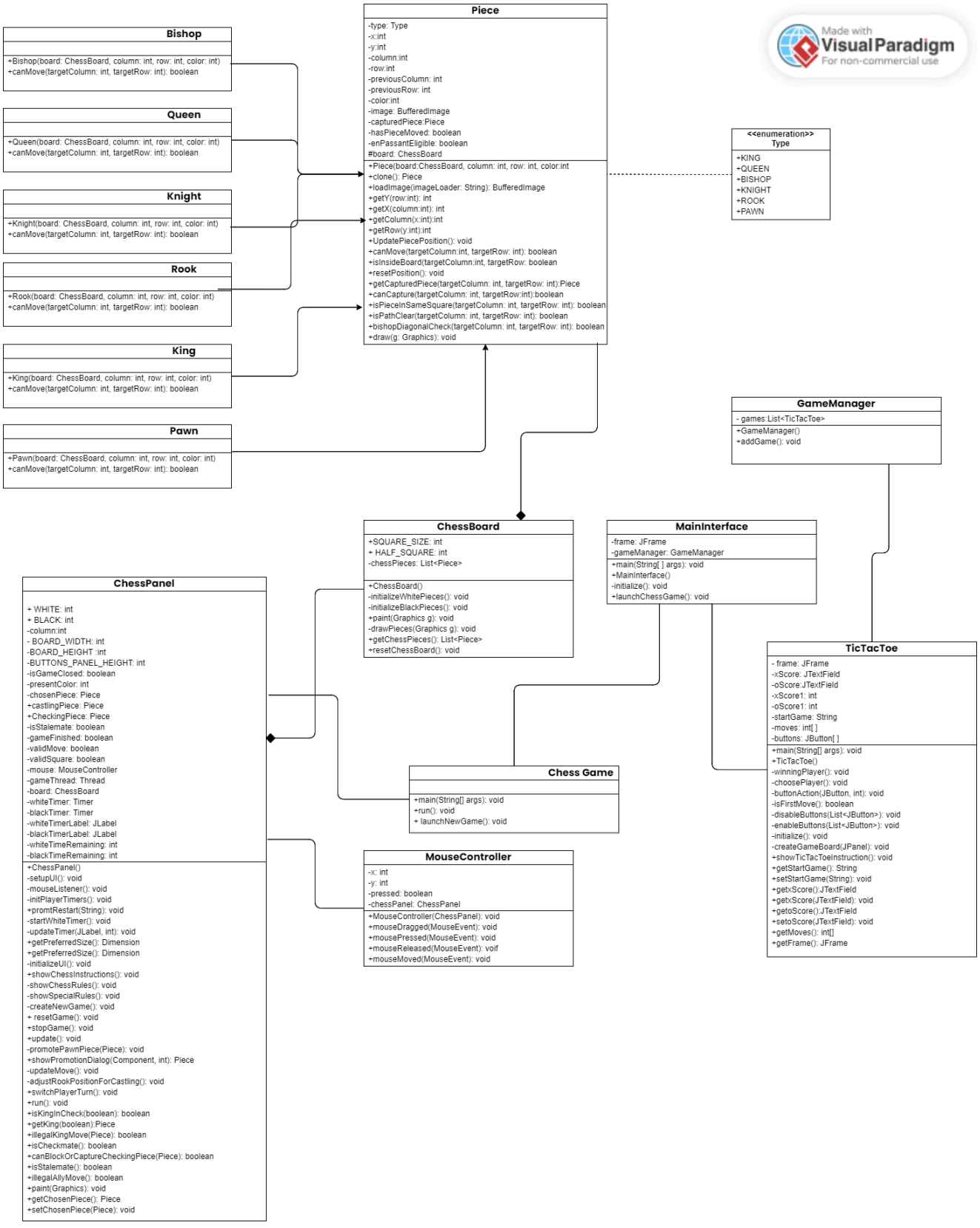
The image above shows the test file for the pawn piece. In the test file above TDD is used for the pawn piece to check for it’s movement and capture logic.

```java
public class KingTest {

    // White Piece Testing
    @Test
    public void testValidMoveWhite() {    // Tests for white Kin

        ChessBoard board = new ChessBoard();
        board.getChessPieces().clear();
        King king = new King(board, 2, 3, ChessPanel.WHITE);
        assertTrue(king.canMove(2, 4));// Verify the king can m
        assertTrue(king.canMove(3, 3));
        assertTrue(king.canMove(2, 2));
        assertTrue(king.canMove(1, 3));
        assertTrue(king.canMove(3, 4));
        assertTrue(king.canMove(1, 2));
        assertTrue(king.canMove(3, 2));
        assertTrue(king.canMove(1, 4));
    }

    @Test
    public void testInvalidMoveWhite() {
        ChessBoard board = new ChessBoard();
        board.getChessPieces().clear();
        King king = new King(board, 5, 5, ChessPanel.WHITE);
        assertFalse(king.canMove(7, 7));
        assertFalse(king.canMove(5, 7));
        assertFalse(king.canMove(7, 5));
    }

    @Test
    public void testInvalidMoveOutsideBoardWhite() {
        ChessBoard board = new ChessBoard();
        board.getChessPieces().clear();
        King king = new King(board, 0, 0, ChessPanel.WHITE);
        assertFalse(king.canMove(-1, 0));
        assertFalse(king.canMove(0, -1));
        assertFalse(king.canMove(8, 0));
    }
```

This KingTest class above illustrates comprehensive testing done using Test-Driven Development (TDD) to validate the functionality of the King piece in a chess game. Each test case verifies every aspect of King's movement and capturing capabilities. A similar approach has been carried out for all the other test files for other pieces.

3.4 UML Class Diagram & User Stories

UML Class Diagram



The above image gives a full overview of my UML class diagram. Since it's blurry a recommendation is to zoom in by firstly opening the 3 dot Google control button and navigating to the magnifier to zoom to at least 200%.

User Stories

Player:

- As a player, I want to be welcomed with a nice and interactive interface.
    - When "MainInterface" is running, the user is welcomed with an interface that presents them with two game options "Tic-Tac-Toe" and "Chess" with images to make GUI more aesthetic.
- As a player, I want to start a tic-tac-toe game so I can play with another player.
    - When the "Tic-Tac-Toe image" panel is clicked in the "MainInterface", a pop-up appears giving the user instructions to play the game when 'ok' is clicked it launches a new tic-tac-toe game window should show up.

- As a player, I want to start a new chess game so that I can play against my friends as an opponent.
    - When the "Chess Image" panel is clicked in the "MainInterface", a pop-up appears that presents the user with chess game instructions and prompts to get user input which is to click 'ok' This launches a new chess game.

- As a player, I want to select how long me and my opponent should have the timers set for.
    - Unfortunately throughout this term, I couldn't finish this feature due to insufficient time and resources. Have I implemented this, I would've allowed the user to select from a range of options starting from 1 minute, 3 minutes, 5 minutes, 10 minutes, and 20 minutes. The standard timer right now is set to 10 minutes meaning each user is allocated 10 minutes and a total of 20 minutes to finish the game.

- As a player, I should be presented with a game mode option to allow me to play against an AI or play two-player chess.

    - Throughout this term even after solid efforts, I could only add two-player mode which is a standard mode. The user can play with another user. Each can allocate themselves the color and white always goes first and so does the timer countdown starting with the white piece first. Due to insufficient time, resources, and knowledge about executing an AI-based opponent, I couldn't add an AI opponent this term. With enough time and resources, I would've given users an option from easy, medium, and hard and would've developed the AI based on the difficulty.

- As a player, I want to be able to make moves in the tic-tac-toe game so I can play the game.
    - The game alternates between 'X' and 'O' players for each move.
    - Clicking the button on the game board should update the board by updating the symbol and color of the button to 'X' or 'O' based on the player's turn.
    - The game logic created should determine a winner or declare a draw.

- As a player, I want to see my score to track my progress.
    - The score should be displayed on the game board and be updated based on the outcome.

- As a player, I want to exit the game so I can quit playing.
    - Clicking the “EXIT” button should prompt a confirmation dialog.
    - If confirmed the entire game window should be closed.

- As a player, I want to reset the game so I can reset the game board to play again.
    - Clicking the “RESET” button clears the tic-tac-toe game board.
    - Scoreboards for players X and O are retained.

- As a player, I should be able to create multiple tic-tac-toe games.
    - Clicking on the “tic-tac-toe image” button should keep creating new tic-tac-toe instances allowing users to create multiple games.
- As a player, I want to move chess pieces so I can feel engaged in the game by following chess rules.
    - Given it’s the user's turn in an ongoing chess game, when the piece is selected with the mouse, and placed in a valid square then the pieces move to the selected square if correct chess rules are followed.
- As a player, I want to capture my opponent's pieces so that I can progress towards winning the game.
    - Given the opponent piece is in a position that the user can legally capture when the player moves to the opponent piece then the piece should be removed from the board and the piece which captured it status on its square.

- As a player, I want to put my opponent in check or checkmate.
    - When the player makes a move that puts the opponent in check, the opponent can either move out the check capture the checking piece, or lose the game if it’s a checkmate.

- As a player I want to perform special chess moves on my opponent which includes castling, en passant, and pawn promotion.

    - Given the conditions of these moves and the rules set for these conditions if they are met the user can perform these special moves.

- As a player, I should be able to create multiple chess games and tic-tac-toe games and play them separately without affecting each other.

    - When the player launches multiple chess games or multiple tic-tac-toe games, the player can play each game simultaneously without affecting the state of other games. This is carried through multi-threading to ensure the player is playing in a concurrent gaming environment.

3.5 Documentation

Throughout the development of my project, I have taken a comprehensive approach to documenting my work. I have utilized effective documentation principles to ensure project clarity, maintainability, and ease of use. This was possible by using documentation methods such as inline comments, JavaDoc, and external files which include README.md, CHANGELOG.md, diary.txt, and a notebook.

In my project, I used inline comments and JavaDoc extensively throughout the codebase to enhance readability and provide useful context to readers in future developers and code maintainers. The approach I took served several purposes.

- **Code clarity:** Ensuring meaningful inline comments were placed in the codebase where clarification of complex methods was required made the code more accessible to new developers.
- **JavaDoc comments:** Using JavaDoc comments in classes and methods provided a greater depth of explanations which included descriptions of methods, parameters, return values, and any expectation thrown or caught. The JavaDoc is useful for any developer looking to extend my project. It's vital to follow this principle to ensure good software engineering principles are followed.
- **External Documentation:** The use of external documentation in my project has allowed me to gather detailed information about the project, setup instructions, version history, and individual efforts towards the project.

    - **README.md:** This markdown file contains setup instructions and how to run the application. This ensures users can quickly get the application running in their local environment. Usage instructions help users navigate and utilize game features to their full potential.

    - **CHANGELOG.md:** This markdown file contains logs of all the changes made across versions including new features, bug fixes, and improvements made. In my project, I made use of this to log all the changes made to my application. This file is vital for users as it contains the development progress of my application from scratch to finish. This will help the user track and understand the impact of updates and facilitate them with debugging and version control if they ever choose to expand on the project.

    - **Diary.txt:** This file contains the personal log of my research conducted throughout Phase 1 and Phase 2 of the project. It contains the development efforts carried through the project lifecycle. I have dated and briefly provided a description of the research and work conducted during the dates. Logging work on the diary has given me time to reflect on my development progress including successful implementation and setbacks faced.

    - **Notebook:** This document contains key notes and information about my project from the discussions I've had during meetings with my supervisor.

4 End Game Development

4.1 Architecture of End System

In the architecture of the end system for my program, particularly the chess game, I have demonstrated a practical application of the Model-View-Controller (MVC) design pattern, ensuring a clear separation between the game's logic, its visual presentation, and the way it interacts with user inputs. This architecture enhances maintainability, scalability, and the ability to modify my application with new features.

Model: Game Logic

The MVC architecture in my chess game is established through the Model components which encapsulate the game's core logic and its state without concern for user interaction. The ‘ChessBoard’ class represents the state of a chessboard. The pieces within the board (Rook, Knight, Bishop, Queen, and King) are all inherited from the abstract class ‘Piece’.

View: User Interface

The View component is shown through the ‘ChessPanel’ class, which is responsible for rendering the game's visual representation based on the current state of the ChessBoard model. It draws the pieces at their current positions and updates the display to reflect the moves made during the game while also providing visual feedback to the user.

Controller: Handling User Input

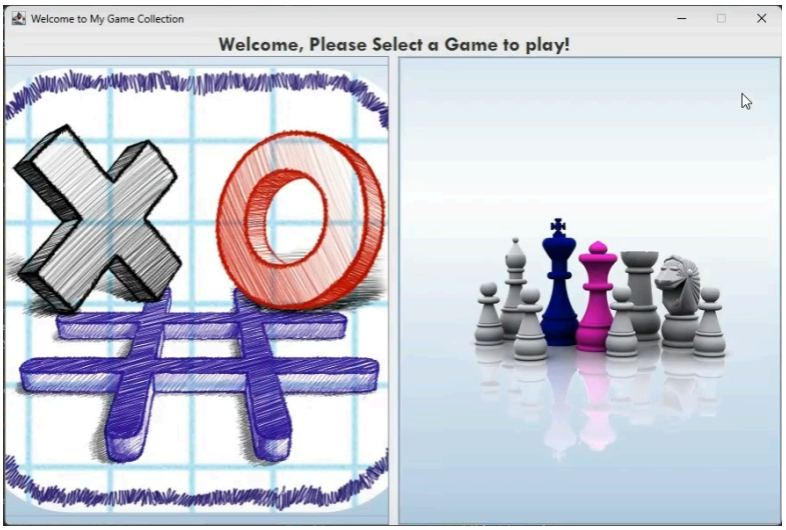
User interactions with the chess game are managed by the Controller component, mainly through the ‘MouseControlle’r class. This class listens for mouse events, such as clicks and drags, and it interprets these actions in the context of the game's rules (Model) and updates the game state accordingly.

While my program supports more than just the chess game, the implementation shown within this section clearly aligns with the MVC architecture. It showcases how dividing an application into individual responsibilities can lead to an application that is easy to maintain and understand. This approach lays a solid foundation for further development, whether by enhancing the chess game itself or by adding new games to my application. This architecture also allows a clear structure for my chess game and also provides a robust framework for extending the application with additional features, following the same MVC principles.

4.2 Features of the End System

This section will highlight all the main features of my application and I will also provide images of the features implemented.

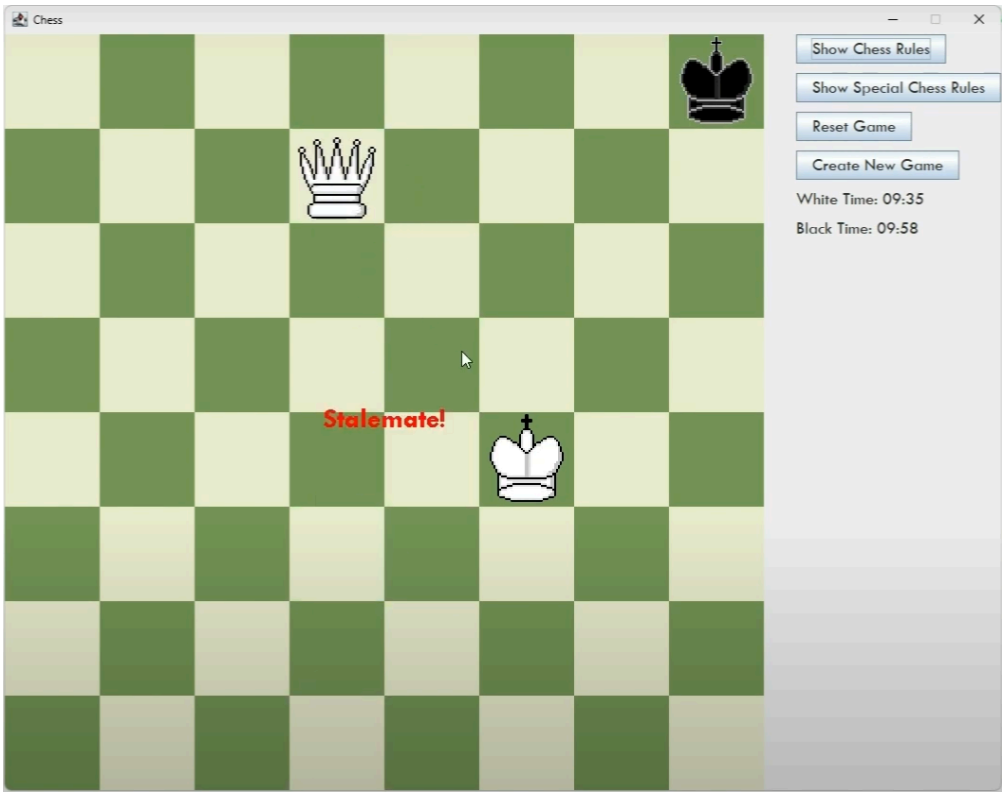
Upon starting the game user is welcomed with an interactive interface. Upon clicking either panel user will launch instances of either game which run on separate threads. The interface below is interesting because it launches separate threads for each game upon clicking on the panels. User



Upon launching the Tic-Tac-Toe game user is welcomed with an instruction box and similarly when launching the chess game user is welcomed with an instruction box containing chess game instructions.

Another interesting feature of my program is the special rules for my chess game. The above chessboard image on the left shows the state of the board before the castling move was carried and on the right, we can see when castling is applied. Castling allows the king pieces to move two spaces to their right or left.

Another special feature and probably the most complex part of my program is the checkmate. In the image above we can see how Checkmate is carried by the black queen piece which results in the white king getting checkmate. This is a win base condition in chess as shown in the image winning message is displayed indicating black piece has won. Checkmate occurs when the king is placed in check and has no legal moves to escape.

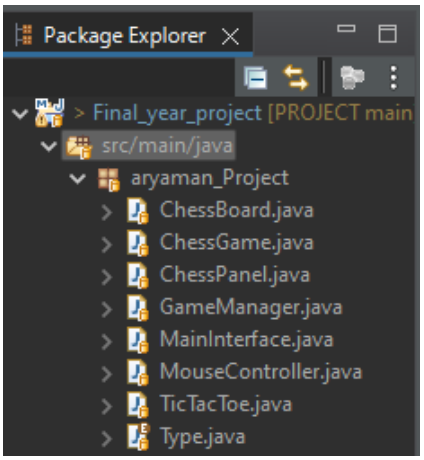


Stalemate is another interesting move in my game which was quite hard to implement. Stalemate occurs when the player whos turn it is to move is not in check and has no legal moves left. This draws the game. In the image above we can see the black king has nowhere to move but is also not in check.This results in a stalemate.

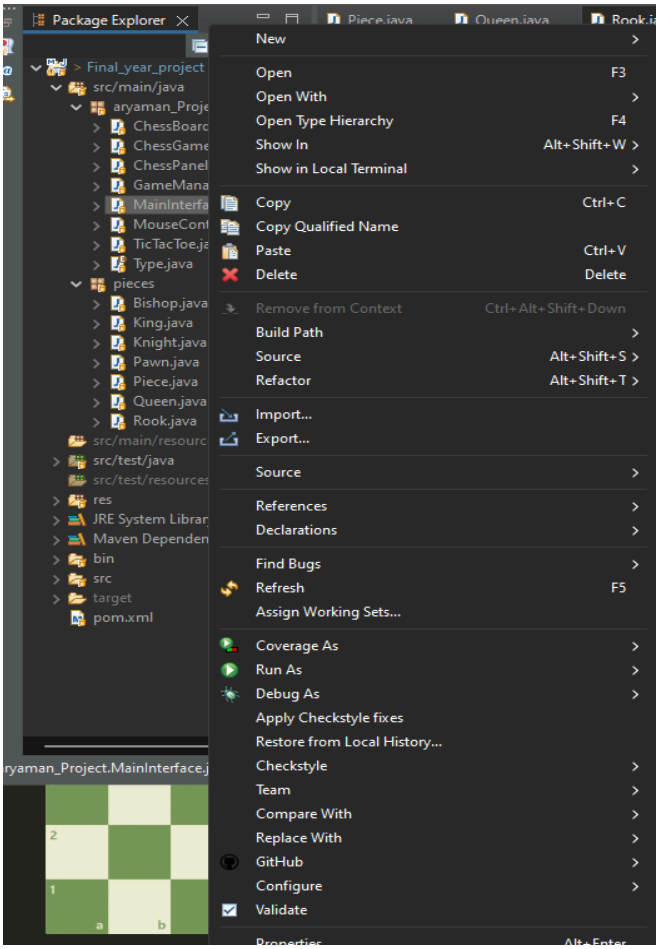
4.3 Running the Application

To run the program the user must follow the below steps:

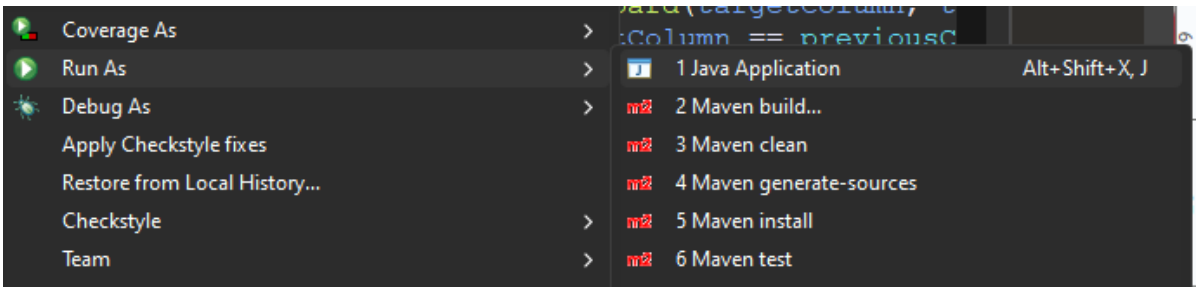
1) Navigate to the Final_year_project folder.



2) Expand the folder src/main/java



3) Find the file “MainInterface.java” and right-click on that file.



4) Next you select ‘Run As’ and click on Java Application

4.4 Potential Future Enhancement

For my trajectory of this project, I would like to further develop my chess game to suit enhancements aimed at elevating user experience. A key area of focus will be placed on adding multiple game modes, where users can choose to engage in matches with AI depending on its chosen difficulty. This addition will not only diversify the game experience but will also accommodate players of all skill levels. Furthermore, I would also like to expand on the time-based options beyond the existing 10-minute limit per user. I would like to allow players to modify game durations to their preference. I would also like to add a history box which is another interesting feature planned for future enhancement of my chess game. The history box should record all moves made during the game, at the end, users should be able to see their moves and enable them to undo/redo moves to facilitate them to learn from their mistakes and learn new techniques they can apply in future games. Lastly, I would like to adopt a more sophisticated concurrency methodology to further improve my application. The focus on refining the thread management system will not only help me achieve technical excellence but also ensure a seamless user experience by keeping the gaming environment robust.

5 Assessment

5.1 Risk And Mitigation

My project, like any other project, comes with its own set of risks and challenges. It’s vital to identify risks earlier in the development stage so they can be managed by ensuring a correct mitigation strategy is followed. In this section, I will discuss common risks and their potential mitigations and then talk more specifically about my project.

Requirement Creep

Expanding the scope of the project or some may say becoming over-ambitious instead of sticking to initial plans can lead to requirement creep which could further delay the project. To mitigate this risk I will define a clear scope of my project and stick to it and prioritise features that are essential and required to complete my project.

Limited Resources

As I will be the sole developer of my project, I have limited time and technical knowledge to complete the project. To mitigate this I will set realistic goals based on available resources and improve my skills as required throughout the project to help me finish it without causing too many problems.

Hardware Failure

There is a possibility the main hardware being used may fail causing major data loss and stunting my progress. Since this is an unavoidable and unpredictable risk I will mitigate this by regularly committing my work in the provided git lab repository to ensure work is stored safely so it's unaffected by hardware failure and I can recover from one quicker.

Overworking On Project

Working without taking regular breaks over a prolonged period of time may result in burnout both mentally and physically and could potentially reduce my productivity and motivation to finish my project. I will mitigate this by simply having a balanced work schedule, taking regular breaks, and not overworking myself. This will allow me to keep a healthy work-life balance which will be beneficial to my mental well-being and thus help me not become stressed out.

Technical Issues

Technical issues may pop up during the development process such as bugs or platform compatibility issues which can be very time-consuming. To mitigate this risk I will ensure I break down my projects into smaller manageable chunks allowing me to regularly carry tests and debug the code. In case there are updates or issues with the current IDE, I will ensure I’m up to date by seeking help from online forums to encounter such technical difficulties.

Concurrent Complexity

As previously researched in the abstract section, achieving concurrency is a challenging task, and managing it can be more complex which could lead to bugs and performance issues. To mitigate this

risk I will ensure I plan the concurrency model carefully and use correct concurrent methods and synchronisation mechanisms before coding. I will also code important features of my project using (TDD) Test-driven development approach so I can test along to resolve any concurrency-related issues.

Performance Bottlenecks

Writing inefficient code can lead to inefficient concurrency which can lead to performance bottlenecks causing reduced game responsiveness. To mitigate this I will make use of profiling tools to help me identify performance bottlenecks. I will also ensure the load on each thread is balanced and not just overloading a single thread.

Deadlocks

Poorly designed concurrent methods can lead to deadlocks where two or more threads are permanently blocked as they are waiting for each other to finish which requires either process to be stopped so the other can finish running. To mitigate this good concurrency code must be written and good implementation of deadlock strategies must be used.

Starvation

Another possible risk is starvation where threads get locked because resources that are required are used by another thread. When threads access resources they enter the “Critical Region” where only one thread can be present at a given time and other threads mustn't enter. To prevent this I will use locks to prevent multiple threads entering critical regions at the same time.

Documentation

Documenting the development of my project is just as important as coding it. Inadequate documentation will result in the project being delayed over time as both coding and documentation are equally important to complete the project successfully. To mitigate this risk I will ensure I leave enough time to complete my reports after my coding. I will also ensure I document all the progress I make on a weekly basis and document design decisions and concurrency strategies throughout to keep my document well organized.

Code Maintenance

Since this is a large project for an individual developer, there is a risk of code confusion over time as more and more lines of code are added to the project which can cause code smells to develop and overall create a poor code structure which is hard to maintain over time. To mitigate this risk I will ensure I keep the code organised by following industry-standard software engineering practices to avoid such issues.

5.2 Professional Issues

In the development of my project, a concurrency-based game environment featuring chess and tic-tac-toe, I encountered several professional issues within my project. While developing Chess and Tic-Tac-Toe, critical aspects of usability, plagiarism, and licensing have come into serious consideration each playing a pivotal role in the project's development. These aspects are instrumental in shaping the ethical framework, user engagement, and legal standings of the project.

During the creation of my concurrency-based game environment with the traditional board games of Tic-Tac-Toe and Chess, I encountered numerous work-related challenges, each with its own set of difficulties and considerations. The primary issue was plagiarism in the creation of this concurrent gaming environment. The growth of this project needed a careful approach when handling intellectual property rights, mainly focusing on plagiarism. It is morally important to acknowledge any third-party code or concepts deployed, to honor and recognize the contributions of countless developers and intellectuals who have developed these games and their online versions. Giving credit where it's due is crucial as it helps avoid legal problems and makes sure the project code is original and correctly referenced. This follows the rules of the ACM Code of Ethics and Professional Conduct [20].

My commitment to contributing to the open-source community was lined up with the Eclipse Public License (EPL), known for its clauses that protect the rights of original authors and promote open-source collaboration. Eclipse was chosen as the development environment not only because of its robust support for Java but also because it goes along with the principles of open-source development [22]. This choice was further reinforced by managing my project on GitLab, a platform that epitomizes the collaborative nature of open-source development. Adding these changes to my project made me realize how licenses affect my project-sharing abilities and future contributions. The understanding came from respecting and appreciating the shared values and teamwork found in other open-source projects. By integrating the EPL, I aimed to follow a development approach that required all modified versions to stay open. This is in line with the principles outlined in the Software Engineering Code of Ethics and Professional Practice [21].

In conclusion, the development of a concurrency-based game environment featuring Chess and Tic-Tac-Toe has been a venture that has been marked by significant professional considerations. Addressing issues related to plagiarism and licensing was not just about meeting project requirements but about embedding the project within a wider ethical and practical context. These considerations highlighted the importance of creating software that is accessible, original, and ethically sound. Through this project, I have sought to navigate these complex issues with care, contributing to the rapidly evolving landscape of digital game development while respecting the work of these classic games. The experience has taught me the sincere impact of professional issues in software development, which has not only helped my final work but also the values it evokes and promotes.

5.3 Self-evaluation

From the start till date, I have gained a substantial amount of skills whilst adhering to software engineering principles. I have learned to integrate a concurrency-based gaming environment into my applications. The journey took me along significant growth and challenges which I had to myself solve. The growing demand for ethical considerations has taught me well to always acknowledge third-party contributors and honor the individuals and intellects that allow us to gain so much information about various topics. I have learned the importance of following a good ethical practice. This has helped me gain legal awareness which was guided by the ACM Code of Ethics and Professional Conduct and the Eclipse Public License's principles.

In this project, I have also increased my technical skills by following TDD methodology which has significantly enhanced my codebase to allow my applications to run with full robustness. The implementation of the MVC framework has shown me the significance of following such architecture to allow organized code which promotes maintainability and scalability. Furthermore understanding effective software development strategies has enriched my understanding and has prepared me well for my future endeavors.

6 Bibliography

[1] A. Walilko, "The Difference Between CPU Cores and Threads," 2023. [Online]. Available: https://www.liquidweb.com/blog/difference-cpu-cores-thread/#:~:text=A%20 thread%20is%20a%20 sequence,speed%20and%20 efficiency%20of%20 multitasking.

[2] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, "Java Concurrency In Practice," Addison Wesley Professional, 2006.

[3] J. Deacon, "Model-View-Controller (MVC) Architecture," 2009. [Online]. Available: http://www.jdl.co.uk/briefings/MVC.

[4] J. Marner, "Evaluating Java for Game Development," 2002. [Online]. Available:http://www. rolemaker.dk/articles/evaljava.

[5] J. W. Cooper, "Java Design Patterns: A Tutorial," Addison Wesley, 2000.

[6] R. Nystrom, "Game Programming Patterns," Genever Benning, 2014.

[7] S. Kant, "Starvation of threads in Java," IEEE, 2020. [Online]. Available: https://medium.com/javarevisited/starvation-of-threads-in-java-e3d6bcfeb770.

[8] Java Platform, Standard Edition (Java SE) Documentation. "Lesson: Concurrency in Swing." [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html.

[9] Oracle Corporation, "Class SwingWorker<T,V>," Java™ Platform, Standard Edition 7 API Specification, Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html.

[10] D. Cohen. 2022. CS2800: Software Engineering(Test Driven Development).

[11] A. A. Cleveland, "The Psychology of Chess and of Learning to Play It," The American Journal of Psychology, vol. 18, no. 3, pp. 269-308, Jul. 1907, University of Illinois Press. [Online]. Available: https://www.jstor.org/stable/pdf/1412592.pdf.

[12] Oracle, "Oracle Documentation." [Online]. Available: https://docs.oracle.com/en/.

[13] Wikipedia contributors, "Rules of chess," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Rules_of_chess.

[14] Chess.com, "Learn Chess." [Online]. Available: https://www.chess.com/learn.

[15] M. J. Coulombe, "Games meet Concurrency: Algorithms and Hardness," Ph.D. dissertation, Dept. Elect. Eng. and Comput. Sci., Massachusetts Inst. of Technology, Cambridge, MA, USA, 2023. [Online]. Available: https://hdl.handle.net/1721.1/151237.

[16] Y. Markushin, "How complex is the game of chess?" TheChessWorld, Jul. 04, 2023. [Online]. Available:https://thechessworld.com/articles/general-information/how-complex-is-the-game-of-chess/.

[17] K. Salen and E. Zimmerman, Rules of Play: Game Design Fundamentals. Cambridge, MA, USA: MIT Press, 2004.

[18] Chess.com, 'How Chess Games Can End: 8 Ways Explained.' [Online]. Available: https://www.chess.com/article/view/how-chess-games-can-end-8-ways-explained#:~:text=There%20are%20three%20main%20ways,%3A%20checkmate%2C%20resignation%20and%20timeout.

[19] D. Radigan, 'An Intro to Agile Project Management,' Atlassian. [Online]. Available: https://www.atlassian.com/agile/project-management/project-management-intro.

[20] ACM, "ACM Code of Ethics and Professional Conduct," Association for Computing Machinery, 2018. [Online]. Available: https://www.acm.org/code-of-ethics.

[21] IEEE Computer Society and ACM, "Software Engineering Code of Ethics and Professional Practice," 1999. [Online]. Available: https://www.computer.org/education/code-of-ethics.

[22] "Wikipedia Contributors, 'Eclipse Public License,' Wikipedia, The Free Encyclopedia, 11 March 2024, [online]. Available: https://en.wikipedia.org/w/index.php?title=Eclipse_Public_License&oldid=1213250329.

7 Planning & Time-scale

Below are the details of milestones that I aim to achieve for my project during the development phases of term 1 and term 2.

By the end of term 1:

- Researched the concept of concurrency-based game environments.
- Create a detailed project plan with an abstract, timeline, risk assessment, and bibliography.
- I finished setting up the project using Java Eclipse with the required debugging/testing tools, libraries, and dependencies.
- Implemented a single working board game with a start and end state.
- Learned and implemented threads on a separate tutorial branch.
- Started implementing AI player but development is on hold until further meeting with the supervisor.
- I have implemented a simple GUI that allows users to create a tic-tac-toe game.
- Implement GUI with buttons for working board games.
- Implemented game logic for tic-tac-toe using Junit test cases following TDD and other SE principles.
- Developed a concurrency-based environment that allows tic-tac-toe games to run on separate threads and moves made by users on its thread.
- Refactoring of code is done and Java doc and comments are added to ensure good coding standards are followed.
- Started GUI work on a chess game.
- Created a user manual on how to run the application inside the README.md file.
- Prepared presentation for my project.
- Created Class diagrams and User Stories.

By the end of term 2:

- Had a meeting with the supervisor to update my project complexity.
- Implemented advanced chess game rules and test them using Junit test cases.
- Implemented chess into my program with complete UI and testing for game logic.
- Implemented threads into chess to achieve concurrency.
- Made the main interface more aesthetic and made it display a chess game option.
- Allowed users to exit or create new instances of tic-tac-toe and chess games.
- Added more JUnit tests to validate new checks and other chess piece rules.
- Debugged, refactored, and evaluated code for better optimization.
- Performed thorough testing on all game instances and GUI and made improvements if any issues came up.
- Finish complete documentation which documents planning, design, implementation, and testing.
- Updated the user manual once the final application is ready for the game environment.
- Created demo to showcase the project.

Acronyms

CPU Central Processing Unit

OO Object Oriented

OOP Object-Oriented Programming

MVC Model-View-Controller

UI User Interface

VM Virtual Machine

IDE Integrated Development Environment

SE Software Engineering

TDD Test Driven Development

EDT Event Dispatch Thread

Appendix

Diary:

02/10/2023 to 03/10/2023:

-Researched different board games and how each board game operates.

-Played chess online with AI and with online players on all difficulties (easy, standard, and hard) to gain a better understanding of chess rules.

04/10/2023 to 07/10/2023:
-Set up my coding environment which is Eclipse(IDE).
-Watched last year's online lecture on Gitlab and YouTube to learn how to use Git commands to properly store our code.

08/10/2023 to 11/10/2023:
-Improved my understanding further of JavaFX which was used on last year's project as well.
-Watched videos on YouTube covering Eclipse shortcuts to improve my coding performance.
-I watched videos on Maven and learned why it's so useful for my project.

12/10/2023:
-Rest from working to avoid overworking.

13/10/2023:
-Researched further into Apache Maven and its advantages in application development.

14/10/2023:
-Installed essential plugins for my Java Project on Eclipse in the Eclipse marketplace. Installed the Checkstyle plugin and FindBugs plugin.

15/10/2023 to 16/10/2023:
-Found existing solutions to developing chess board games and naughts and crosses and planned out how I could improve them and use them for my project.

16/10/2023 to 25/10/2023:
-Researched WindowsBuilder and Swing design as alternatives to JavaFX.
-Installed WindowsBuilder to use Swing design for my project.
I started creating a GUI for my TicTacToe game.

25/10/2023 to 07/11/2023:
-I finished the GUI for my TicTacToe game.
-Started coding the game logic for my game using the TDD approach making commits almost every day.
-Started updating my interim report.

07/11/2023 to 16//11/2023:
-Competition of tic-tac-toe board game.
-Completion of all possible tests for tic-tac-toe.
-Removed code smells from files.
-Added Java documentation and comments.
-Started playing chess to learn its game logic.
-Cleaned up the repository by removing unwanted files.

16//11/2023 to 27/11/2023:
-Learned about threads/multi-threads.
-Started implementing Chess game.
-Re-factored my tic-tac-toe game before implementing threads.
-Updated my interim report.

27/11/2023 to 01/12/2023:
-I did some minor refactoring.
-Updated interim report.
-Implemented thread to my tic-tac-toe game.
-Tested thread implementation.
-Added Java doc and comments.

01/12/2023 to 06/12/2023:
-Fixed broken test cases.
-Updated readme.md file which includes how the app works.
-Fixed exit button problem.
-Updated interim report.

15/01/2024 - 24/01/2024:
-Re-visited chess board game and researched further into the game rules, design, movements for each piece, and checkmate aspect of chess game.
-Finished designing the chessboard and its pieces.
-Cleaned up the repository by removing unwanted files.
-Further research going on to start implementing pieces so it can move around the board.

24/01/2024 - 01/02/24:
-Started final year report
-Researched interfaces that will be required for chess pieces to move
-Implemented interfaces for pieces to move when the user uses the mouse.
- Finished basic chess piece movement without rules
- Started Unit testing on the movement of each piece according to capture/movement rules to develop each piece.

01/02/24 - 09/02/24:
-Resoved image rendering issue.
-Updated final year report.
-Implemented movement functionality on different pieces and added new capturing methods for each piece.

09/02/24 - 20/02/24:
-Updated final year report.
-Finished movement and capture functionality for each piece.
-Finished testing for white chess pieces.

20/02/24 - 27/02/24:
-Updated final year report.
-Implemented turn-switching logic for my chess game.
-Finished black piece testing.

27/02/24 - 03/03/2024:
-Updated final year report.
-Started researching checkmate detection systems using online resources.

04/03/24 - 11/03/2024:
-Reviewed work with supervisor so far and tackled any issues/questions I had during online meeting.
-Researched special chess moves promotion, castling, and en passant.
-Finished research on promotion and implemented it in my game.
-Improved GUI and user feedback feature.
-Removed code smells and added useful comments and Java docs to all code files.

12/03/24 - 13/03/2024:
-Reviewed Oracle documentation to learn how to get an image icon inside the options dialog.
-Added images to replace previous text-based promotion dialog options.

13/03/24 - 15/03/2024:
-Added game title inside chess panel and modified player turn switch messages for improved user game experience.
-Further researched deep into En Passant deployment into my game.
-Began En Passant implementation for my chess game.

16/03/24 - 19/03/2024:
-Finished En Passant feature and began casting feature.
-Implemented castling feature.
-Changed chessboard color for better aesthetics.

21/03/24 - 22/03/2024:
-Removed static variables and changed them to instance variables to match concurrency needs.
-Temporarily removed castling-related code.
-Added a new button for launching chess games inside MainInterface class.
-Fixed minor game bugs which were causing issues when running and closing windows.
-Updated all testing files to match new changes.
-Added casting logic to the game.
-Updated report

25/03/24 - 26/03/2024:
-Implemented illegal move check for King piece using TDD.
-Updated GUI to let users see when King is in danger.
-Improved main interface by adding images in place of text for better aesthetics.
-Added instructional pop-ups for both games to allow users to understand how to play.
-Added instructional buttons within the panel to allow users to open them anytime they want to check the rules.
-Carried further GUI enhancements.
-Updated report.

26/03/2024 - 29/03/2024
-Added new buttons to reset the game, and create a new game.
-Implemented check logic & Illegal moves.
-Added timers for white and black users.
-Refactored ChessPanel class.

29/03/2024 - 02/04/2024
-Researched checkmate, stalemate, and other draw-based conditions.
-Implemented checkmate detection mechanism.
-Started stalemate detection.
-Updated report.

02/04/2024 - 04/04/2024
-Implemented stalemate logic.
-Fixed issue with timers.
-Updated report.

04/04/2024 - 05/04/2024
-Added Javadoc comments and inline comments to all test files.
-Finished report.
-Cleaned repository.
-Minor codebase cleaning by getting rid of unused imports.
-Updated README.md file.

Notebook:

Meeting date: 2nd October 2023

- Briefly went over the project details with the supervisor and discussed how to come up with the project plan.
- The supervisor explained to me to put a strong emphasis on exploring various board games for the project. Decided on tic-tac-toe for developing basic foundational concurrent applications.
- The supervisor also recommended developing board games like chess or similar. Which revolves around a concurrency-based game environment.

Meeting date: 26th October 2023

- Reviewed Eclipse setup and initial Git practice. My supervisor suggested me to use Eclipse plugins to enhance productivity. He also told me to further explore WindowsBuilder for Swing application development.
- We further discussed potential challenges in Git Lab usage for code management. My supervisor suggested me to keep the git repository clean and organised to ensure good software engineering principles are followed.
- Additionally, we also discussed deliverables for the interim report.

Meeting date: 23rd November 2023

- In this meeting supervisor provided me further guidance on implementing chess as tic-tac-toe itself wasn't complex enough. Taking this into consideration I decided to begin researching chess straight way while still completing my tic-tac-toe implementation.
- Discussed the project scope complexity in detail with my supervisor and took useful tips on how to submit a strong interim report.
- All the above points were discussed via email due to connectivity issues whilst joining the meeting.

Meeting date: 25th January 2024

- Taking previous meeting notes into consideration, I began the development of my chess game. My supervisor suggested diving deep into game logic for chess and how each of the pieces moves.
- We discussed how to make further progress with the project to ensure all deliverables are submitted on time.
- We also discussed how to manage time better especially when all the deadlines for other modules were catching up with me.

Meeting date: 7th March 2024

- Discussion regarding the final year report and content that needs to go in the report.
- Discuss current progress with the chess game.