Final Year Project Report

Full Unit – Final Report

Concurrency based game environment

Mohammed Zuhayr Mohsen

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Zhaohui Luo

Department of Computer Science

Royal Holloway, University of London

Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 11411(Excluding title page, declaration page, table of contents and appendix)

Student Name: Mohammed Zuhayr Mohsen

Date of Submission: 12/04/2024

Signature: Mohammed Zuhayr Mohsen

Table of Contents

Link to project walkthrough: https://youtu.be/0bJwtWIem2w?feature=shared

Abstract

This report focuses on my implementation of concurrency in a game playing environment in which multiple game threads are able to execute at once. This report will dive into both theoretical concepts of concurrency which have been thoroughly researched and also practical elements that have been implemented in my final system. My approach to achieving concurrency makes use of techniques such as multithreading and synchronization mechanisms in order to create a thread-safe game environment. My project combines concurrency functionality in a graphical user interface which displays the concurrent game playing environment to the user/s.

This report will start out by describing the background theory behind concurrency while also setting out my project goals. It will then talk about the software engineering techniques used in development. Next, it will go into in depth analysis of my software solution while also providing critical analysis on the effectiveness of my software. Finally, there will be a chapter talking about professional issues.

My game of choice is Chess and therefore this project is based on a concurrent chess game environment.

Chapter 1: **Introduction**

Concurrency is a concept in which multiple tasks can be executed at the same time within a system. This is done by splitting processes into semi-processes known as threads [1]. Threads are the most essential concept to grasp when it comes to concurrency because concurrency is achieved via the use of threads. A thread is a small sequence of instructions that can be executed by a processor. Each thread is made up of its own unique stack, program counter, and set of registers [2]. The stack in each thread contains runtime data such as local variables. This allows different threads executing the same instruction to not always return the same value (if local variables are different or manipulated differently). Furthermore, threads share virtual address space with other threads in order to efficiently share data and information between themselves. This therefore makes threads of a particular process dependent on each other due to the shared code, data and files between them; if one thread in a process fails then it is likely the other threads will fail or produce an incorrect result upon execution.

Threads have many advantages in comparison to processes with the main advantage being the ability of multiple threads executing at the same time. This paired with the swift inter-thread communication allow threads to efficiently execute instructions. Furthermore, due to the dependent nature of threads, it allows them to assist each other [3] in operation and therefore improves application performance. Context switching within threads is also very quick due to the shared virtual memory allowing the state of each thread to be rapidly stored and restored by the CPU. This is particularly useful in systems with a vast range of functionality.

A processor that is able to execute multiple threads at once is known as a multithreaded processor. The first multithreaded processors were launched in the late 1900s and were able to solve the memory access latency problem [4] due to the concurrent nature allowing a quicker access to resources. In modern day technology, processors support a concept known as hyperthreading. This concept allows a physical core of a processor to be divided into two virtual cores [5] which therefore allows the processor to work on multiple threads at the same time. As a result, the processor is able to effectively utilize available resources and process performance is very efficient. This shows the effective nature of multithreading in a system.

1.1 The Problem

When trying to implement concurrency in a system, there are various potential issues that may require suppressing when it comes to concurrent programming. As previously mentioned, threads contain shared resources. This creates the possibility of data corruption [6] when these resources are used on multiple threads due to the non-atomic nature of the operations used on this shared data. As a result, the outcome of a multithreaded system could produce inconsistent results. The main reason for this potential data corruption is due to a lack of thread-safety when programming a concurrent system. The potential inconsistency and requirement of thread-safe methodology within a multithreaded system is generally why it is considered harder than single-threaded programming [7]. However, once successfully implemented, a multithreaded system is much more efficient.

There are many different ways of preventing concurrency issues in a system. The first method is by making use of synchronization. Synchronization can be achieved in many ways and if successfully implemented, it allows coordination in multiple threads when it comes to accessing shared program states [8].

1.1.1 Synchronization using locks

One way synchronization can be achieved is through the use of locks. Locks are variables which hold the state of the lock [8] at any given time; this could either be “locked” or “unlocked”. When a lock is in the “locked” state, it means a thread is currently holding onto it and is most likely in a critical section. A critical section is another way of referring to the part of the code in which shared resources are accessed. Subsequently, the “unlocked” state means the lock is available and is not currently held by any thread. These lock variables are usually joined with the methods “lock()” and “unlock()”. Calling the “lock()” method will make the current thread attempt to obtain the lock. This can only be achieved if the lock is vacant(no other thread is currently holding the lock). If this method is successful, the thread is able to acquire the lock and enter the critical section. This allows only a singular thread to enter the critical section at once due to only one thread being able to lock a variable at once. If another thread did call the “lock()” method while the lock was not vacant, they will have to wait until the lock eventually becomes available. This will only happen once the current holder of the lock calls the “unlock()” method which causes the thread to exit the critical section and vacates the lock if no other threads are currently waiting for the lock. Locks are very effective in reducing potential data corruption due to them creating mutual exclusion between threads(only one thread can be in the critical section at a single time). The following lock algorithm described was a very simple algorithm commonly referred to as a mutex lock algorithm. Mutexes are very simple to implement and use, however they require a lot of context switching which can be very costly in terms of processor performance [9]. Although lock algorithms are able to avoid race conditions, they can create a problem in the name of a deadlock. A deadlock is when two separate threads lock a different variable at the same time but then try to lock the variable currently locked by the subsequent thread [10]. This is a severe issue because it results in both Threads remaining in their “waiting” phase as they are waiting for the currently locked variable to become unlocked. However, since both threads are holding the variable the other thread requires, both threads remain in their wait state and nothing is able to happen. Deadlocks can be prevented within a lock algorithm by making use of lock ordering [11]. Putting an order on the locks that are required simultaneously and making the code follow that order can remove the possibility of a deadlock due to it never allowing different threads requiring the same variable at once. For example, in a system that contains two threads, thread A could require variable x and variable y and thread B could require variable y and then variable x. The algorithm will instruct the program to only allow thread A to have access to variable y if it already holds variable x. This removes the chance of thread B locking the variable while thread A is holding variable x. Hence, it removes a deadlock situation. However, this approach is almost impossible to apply in a real world context and therefore is not commonly used. The other lock based solution to the deadlock problem is making use of a Coarse-Grained lock [12]. This type of lock is used to cover many objects in a system which prevents the possibility of a deadlock. However, it can create a huge downgrade in system performance.

1.1.2 Synchronization using a singleton

A very common practical approach to creating synchronization is by making use of thread-safe singletons. The singleton design pattern is used to ensure a class only has one instance in the program. The following is an example of a thread-safe singleton created in the java programming language [13]:

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {
  if (instance == null) {
    synchronized (ThreadSafeSingleton.class) {
      if (instance == null) {
        instance = new ThreadSafeSingleton();
      }
    }
  }
  return instance;
}
```

This algorithm achieves thread safety because it ensures only one instance of the class is created with use of the “synchronized” key word. This also allows only one thread to access the class at once.

1.1.3 The Bill Pugh Singleton

The most widely used approach to achieving a thread-safe singleton was created by Bill Pugh. This was due to it not requiring any synchronization and instead using an inner static helper class [13]. It also provided programmers simplicity in creating a thread-safe singleton. The following is an example of Bill Pugh's singleton algorithm [13]:

```
public class BillPughSingleton {

  private BillPughSingleton(){}

  private static class SingletonHelper {

    private    static    final    BillPughSingleton    INSTANCE    =    new
BillPughSingleton();

  }

  public static BillPughSingleton getInstance() {

    return SingletonHelper.INSTANCE;

  }

}
```

The 'SingletonHelper' cinner class is only loaded once the 'getInstance()' method is run. Therefore, an instance of the main class is only loaded into memory when it is called. This achieves thread-safety due to the algorithm ensuring only one instance of the main class is created and it only being available to one thread at any time.

1.1.4 Concurrency in a game environment

Ensuring thread-safety is achieved is crucial when making use of concurrent programming because it allows the system to benefit from the numerous advantages of multithreading. One main architecture that would really benefit from concurrency is a game environment. A lack of concurrency in a game environment would not allow multiple games to be played at once. In a single threaded game environment, multiple games may be able to coexist, but only one game will be accessible at a time. This defeats the purpose of containing multiple games in a single environment and would therefore strongly encourage the use of multithreading in a game environment. In a multithreaded system, each game could be represented as a Thread and therefore, while running, the environment will be able to switch between the games when told to. This hugely increases the functionality and efficiency of the game environment while also allowing numerous players to play concurrently.

1.2 Project Goals

The main goal for this project is to successfully implement a concurrency based game environment in Java with the help of multithreading. My game of choice is Chess and therefore, I would like this environment to present users with multiple Chess games and allow them to take part in multiple games at once.

1.2.1 Chess logic

One of the most important goals of this project is to successfully implement the rules of chess for each particular game. Every game should support movement logic for each piece, capturing logic, pawn promotion logic, check logic, checkmate logic and also stalemate logic. Incorporating all of this functionality will provide each player with a fully functioning game of chess to play.

1.2.2 Multithreading

This environment would require the use of multithreading; each game of chess would need to be represented as a thread and each game would need to share the same module. In the system, there will need to be a module that each game thread would need to access at some point in the system. In this context, the most appropriate module would be a move legality module in which moves made by users, on any game in the environment, will be checked and verified. This would create a concurrent nature due to multiple moves having the ability to execute simultaneously in the system. This will only be achieved by making the module thread-safe and preventing any possible issues. For this reason, it will be appropriate to set this module up using the Bill Pugh singleton design as it is the most widely accepted singleton design and prevents a lot of concurrency issues.

1.2.3 Graphical User Interface

The game environment will need to have a nice graphical user interface which presents each user with the various games of chess on their screens. The interface should be very consistent but not too overwhelming for users despite the possibility of multiple games being on screen to look at. One way this could be implemented is by making use of a horizontal/vertical scroll bar which can be used to scroll through the different games in the environment. Furthermore, the system should provide appropriate error reports graphically to each user based on erroneous input. These error reports should allow the user to recognize the problem and allow them to fix it. The interface should also provide a status bar for each particular game thread so players can easily distinguish between the status of each game. The game status interface can visually show each player their move history and the number of pieces they currently have.

1.3 Personal Goals

Having previously studied concurrency and multithreading, I have always wanted to apply my theoretical knowledge in a practical sense due to the potential power of concurrent programming. This practical use of concurrency will provide a challenge due to the potential issues that may occur in concurrent programming requiring me to think of ways to suppress them.

I also wanted to use this project as another opportunity to improve my use of version control software, which in this case is git. The use of git in creating this project will further familiarize myself with the methods behind collaborative software creation due to branching systems being used in group software development. By getting myself used to committing changes to a remote repository, it will get me ready for developing software in the real world for real clients. This project also allows me to further practice the use of feature branches for implementing new features and release branches for creating major releases of software.

Another personal goal I have for this project is to explore the subject of game creation. This is something I have always been interested in and I believe a game of chess is a good place to start due to it not being too complex but also not too easy. Furthermore, the rules are well known and therefore testing and debugging will be very easy due to having the ideal functionality already known. After completing the project, it will provide me with a solid foundation when it comes to game creation and I will be able to use this to create further gaming projects which slowly increase in difficulty.

1.4 Literature Review

I previously referred to an online article with the name: “Process vs. Thread” (reference number 1 in my bibliography). This article allowed me to grasp the various differences between threads and processes. It also allowed me to fully understand why the use of threads can be a lot more beneficial to the execution of instructions in a system which therefore spearheaded my decision of making use of threads in my project.

I previously referred to an online literature with the name: “Threads in Operating System” (reference number 2 in my bibliography). This piece of literature provided me with the knowledge of the components of a thread and also described why threads are important and how effective multithreading is. This online literature also provided me with very useful images showcasing the differences between single-threaded processes multithreaded processes. Having the ability to see this visually really helped me grasp the key concepts in multithreading.

I previously referred to an online tutorial with the name: “Multithreading in Java - Everything you MUST Know” (reference number 6 in my bibliography). This tutorial was crucial for me to understand how to practically implement multithreading in a java environment and it allowed me to successfully introduce multithreading in my project.

I previously referred to an online literature with the name: “Reading 23: Locks and Synchronization” (reference number 11 in my bibliography). This piece of literature introduced me to a way of getting rid of deadlocks because lock ordering is a very effective theoretical way of doing so. However, this literature is not one of the more crucial pieces due to lock ordering being very unlikely to succeed in a practical scenario. For this reason, I was unable to use lock ordering in my project but it was still a very interesting and effective theoretical concept to learn about and it helped me to think of other potential ways to avoid synchronization issues.

I previously referred to an online tutorial with the name: “Java Singleton Design Pattern Best Practices with Examples” (reference number 13 in my bibliography). This tutorial allowed me to understand the different practical approaches to singleton design in java. Since I wanted to make use of a thread-safe singleton in my project, this tutorial allowed me to explore the various options and the advantages of each. This allowed me to successfully pursue with the Bill Pugh singleton design in my source code. This literature was arguably one of the most crucial pieces I came across in my research because my concurrent programming approach was directly inspired by this piece of literature.

All of the literature which has been sourced in this report has helped in the creation of this project, whether it has provided me with theoretical knowledge or practical based solutions. The majority of theory based literature used in this report was used to introduce me to new concepts and help me understand the various advantages and disadvantages of different methodologies used in concurrent programming. This knowledge of background theory allowed me to move onto the practical based literature and allowed me to get a hands on tutorial of concurrent programming.

Chapter 2: **Software Engineering**

In this section I will dive into the software engineering techniques and methodologies that were used in my project and the importance of using each one. I will also describe design features in my source code.

2.1 Singleton design pattern

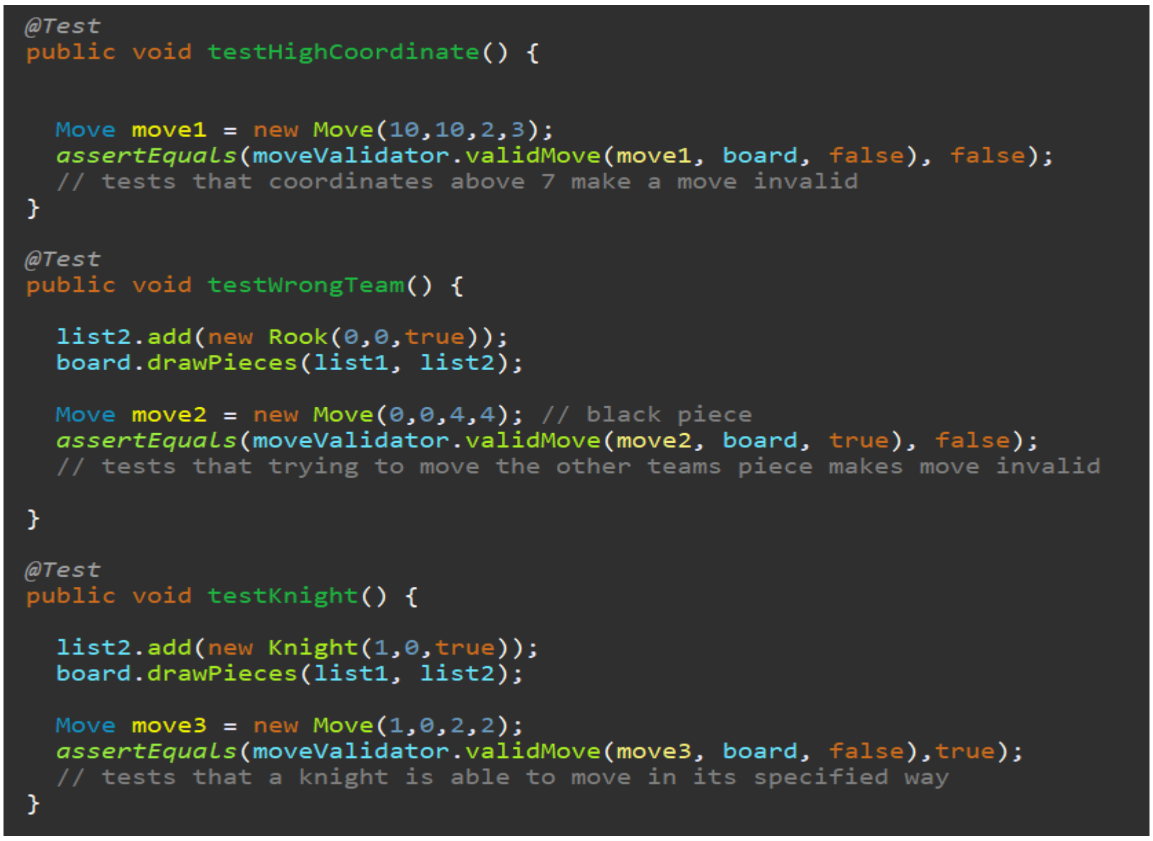
As previously mentioned, in this project I make use of a singleton class. This class being the move validator which is set up using the Bill Pugh singleton. The singleton design pattern is very beneficial if you only want one instance of a specific class in the whole system. This allows shared resources to be managed in one place without the possibility of data corruption.

2.2 Agile development

I followed the agile development methodology in the production of my project. This methodology is an iterative approach to developing software in which functionality is broken down into smaller blocks and implemented one block at a time. This reduces risks when it comes to introducing new software and also allows easy debugging of new functionality. Evidence of this development methodology being used in my project can be seen in my GitLab project repository in which feature branches are created for each new feature and functionality is incrementally introduced via commits to the feature branch. Branches are merged back into main and then deleted once a new functionality is fully added to the system.

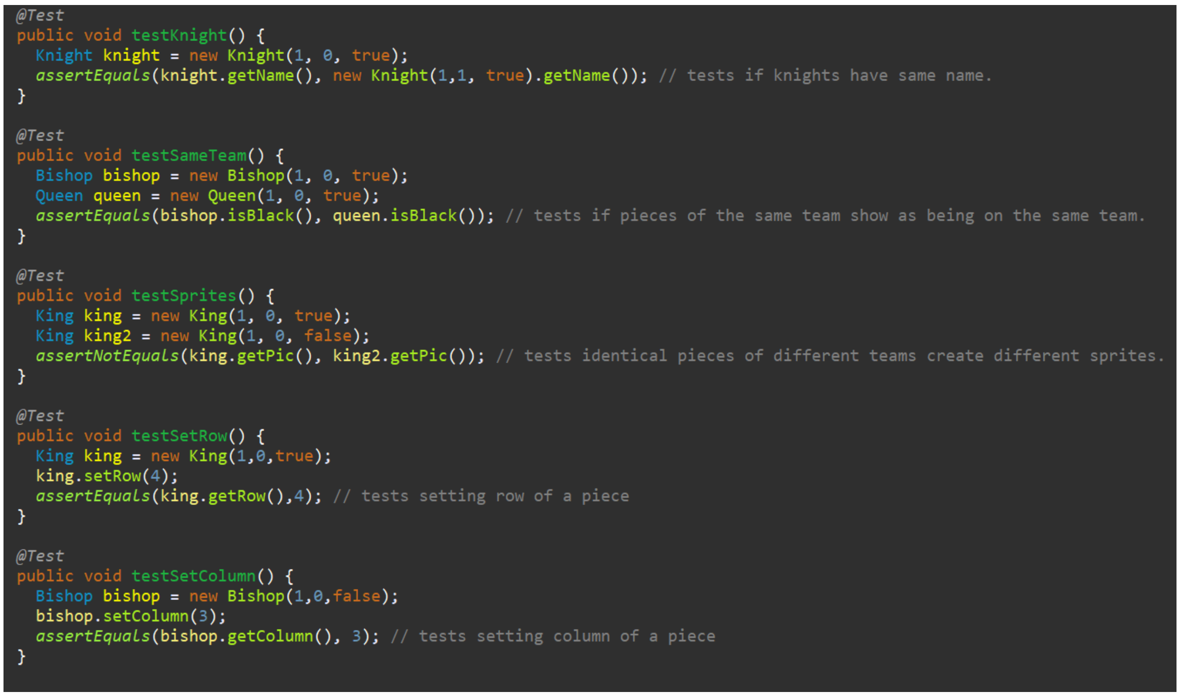
2.3 Test driven development

I followed a test driven development methodology in the process of developing my project. This methodology is an iterative approach to developing software in which an initial failing test case is written and then the code which allows this test case to pass is then written and refactored. This constant iterative cycle ensures functionality is always met and provides a very good approach to implementing functionality. My test cases were written in the appropriate TestX method (where X is the name of the class being tested) and they made use of Junit tests. Below are some examples of tests carried out in my project:



```java
@Test
public void testHighCoordinate() {

   Move move1 = new Move(10,10,2,3);
   assertEquals(moveValidator.validMove(move1, board, false), false);
   // tests that coordinates above 7 make a move invalid
}

@Test
public void testWrongTeam() {

   list2.add(new Rook(0,0,true));
   board.drawPieces(list1, list2);

   Move move2 = new Move(0,0,4,4); // black piece
   assertEquals(moveValidator.validMove(move2, board, true), false);
   // tests that trying to move the other teams piece makes move invalid

}

@Test
public void testKnight() {

   list2.add(new Knight(1,0,true));
   board.drawPieces(list1, list2);

   Move move3 = new Move(1,0,2,2);
   assertEquals(moveValidator.validMove(move3, board, false),true);
   // tests that a knight is able to move in its specified way
}
```

Figure 1. Test cases in TestMoveValidator



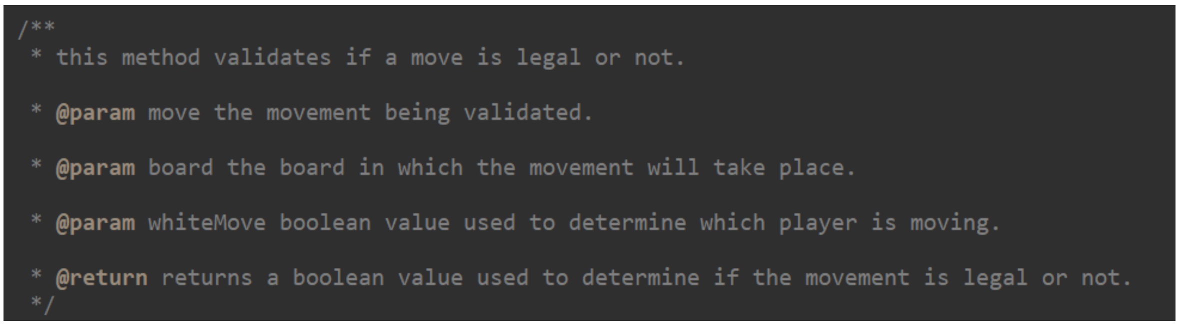
```java
@Test
public void testKnight() {
  Knight knight = new Knight(1, 0, true);
  assertEquals(knight.getName(), new Knight(1,1, true).getName()); // tests if knights have same name.
}

@Test
public void testSameTeam() {
  Bishop bishop = new Bishop(1, 0, true);
  Queen queen = new Queen(1, 0, true);
  assertEquals(bishop.isBlack(), queen.isBlack()); // tests if pieces of the same team show as being on the same team.
}

@Test
public void testSprites() {
  King king = new King(1, 0, true);
  King king2 = new King(1, 0, false);
  assertNotEquals(king.getPic(), king2.getPic()); // tests identical pieces of different teams create different sprites.
}

@Test
public void testSetRow() {
  King king = new King(1,0,true);
  king.setRow(4);
  assertEquals(king.getRow(),4); // tests setting row of a piece
}

@Test
public void testSetColumn() {
  Bishop bishop = new Bishop(1,0,false);
  bishop.setColumn(3);
  assertEquals(bishop.getColumn(), 3); // tests setting column of a piece
}
```

Figure 2. Test cases in TestPieces

Figures 1 and 2 show some Junit 4 tests that were done in my system. The assert equals method ensures the first value given is equivalent to the second value give. For example, in Figure 2, there is a test case called testSetRow and in which, the assertEquals method tests if the row value of the king piece is equivalent to 4. In this case, that is true and the test will pass.

2.4 Documentation

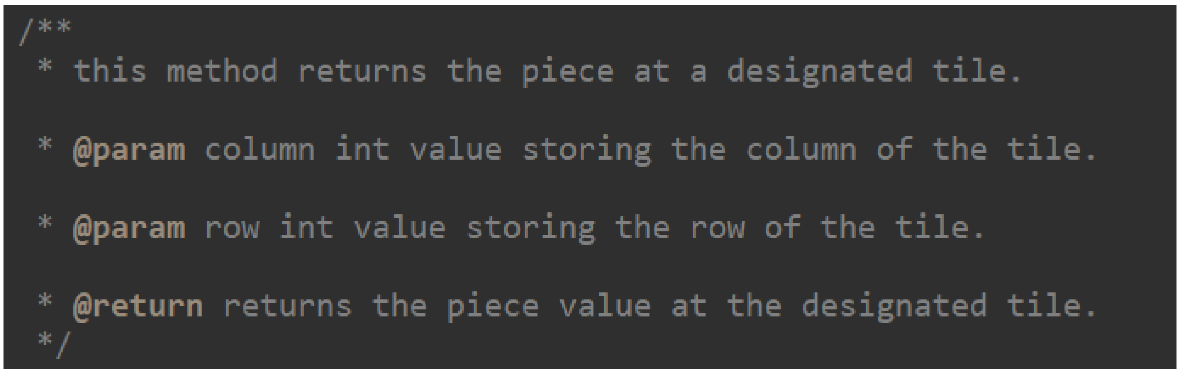
I made use of Javadoc in my project in order to provide an explanation of the different methodology while also making code easier to understand. I had Javadoc for each class and its methods. The following screenshots show some of the Javadoc used in my project:



```
/**
 * this method validates if a move is legal or not.

 * @param move the movement being validated.

 * @param board the board in which the movement will take place.

 * @param whiteMove boolean value used to determine which player is moving.

 * @return returns a boolean value used to determine if the movement is legal or not.
 */
```

Figure 3. Javadoc for valid move method in move validator

Figure 3 shows an example of Javadoc in my system. The first line is used to describe what the method does. The second line describes what the parameter called move represents. The third line describes what the parameter called board represents. The fourth line describes what the parameter called whiteMove represents. The final line describes what the return value of the method represents.



```
/**
 * this method returns the piece at a designated tile.

 * @param column int value storing the column of the tile.

 * @param row int value storing the row of the tile.

 * @return returns the piece value at the designated tile.
 */
```

Figure 4. Javadoc for getPiece method in Board class

Figure 4 shows another example of Javadoc in my system. The first line is once again used to describe the method and the others are used to describe the parameters and the return value of the method.

2.5 Code styling

I made use of a check style in the process of developing my project. My check style was set to the google check style and therefore made me style my code in accordance with the styling rules used in google. One example of a styling rule was a singular line of code never exceeding 100 characters. In my source code you will see that no singular line of code exceeds 100 characters due to this rule. The use of this check style made my code a lot more readable and also very consistent.

2.6 Class design

2.6.1 UML

The following diagram represents the class design and dependencies in my project. Note: if the diagram is too small, please zoom in because formatting issues did not allow me to make the image any larger.

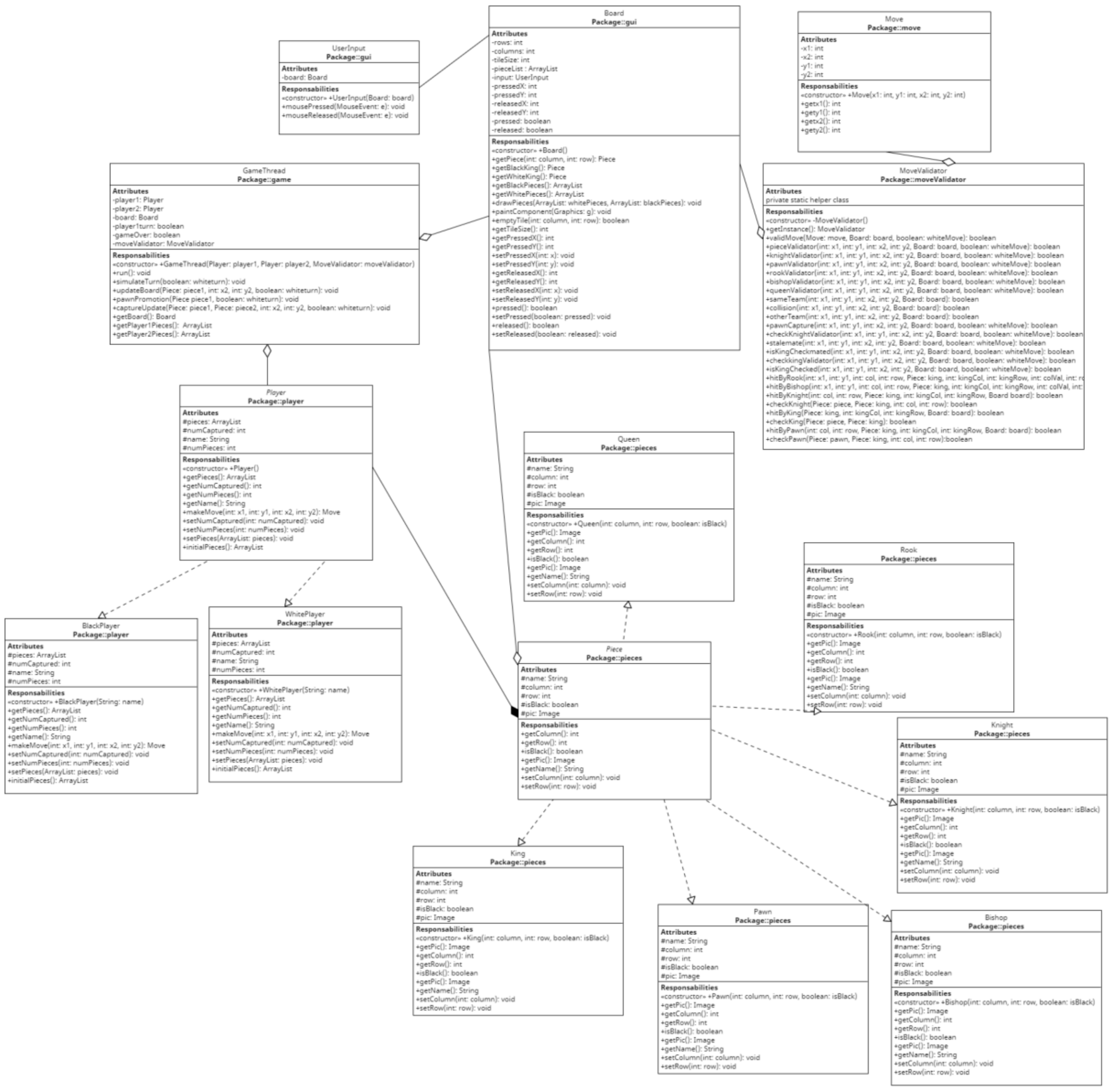

UserInput
Package::gui
Attributes
-board: Board
Responsabilities
«constructor» +UserInput(Board: board)
+mousePressed(MouseEvent: e): void
+mouseReleased(MouseEvent: e): void
Board
Package::gui
Attributes
-rows: int
-columns: int
-tileSize: int
-pieceList : ArrayList
-input: UserInput
-pressedX: int
-pressedY: int
-releasedX: int
-releasedY: int
-pressed: boolean
-released: boolean
Responsabilities
«constructor» +Board()
+getPiece(int: column, int: row): Piece
+getBlackKing(): Piece
+getWhiteKing(): Piece
+getBlackPieces(): ArrayList
+getWhitePieces(): ArrayList
+drawPieces(ArrayList: whitePieces, ArrayList: blackPieces): void
+paintComponent(Graphics: g): void
+emptyTile(int: column, int: row): boolean
+getTileSize(): int
+getPressedX(): int
+getPressedY(): int
+setPressedX(int: x): void
+setPressedY(int: y): void
+getReleasedX(): int
+getReleasedY(): int
+setReleasedX(int: x): void
+setReleasedY(int: y): void
+pressed(): boolean
+setPressed(boolean: pressed): void
+released(): boolean
+setReleased(boolean: released): void
Move
Package::move
Attributes
-x1: int
-x2: int
-y1: int
-y2: int
Responsabilities
«constructor» +Move(x1: int, y1: int, x2: int, y2: int)
+getx1(): int
+gety1(): int
+getx2(): int
+gety2(): int
GameThread
Package::game
Attributes
-player1: Player
-player2: Player
-board: Board
-player1turn: boolean
-gameOver: boolean
-moveValidator: MoveValidator
Responsabilities
«constructor» +GameThread(Player: player1, Player: player2, MoveValidator: moveValidator)
+run(): void
+simulateTurn(boolean: whiteturn): void
+updateBoard(Piece: piece1, int: x2, int: y2, boolean: whiteturn): void
+pawnPromotion(Piece piece1, boolean: whiteturn): void
+captureUpdate(Piece: piece1, Piece: piece2, int: x2, int: y2, boolean: whiteturn): void
+getBoard(): Board
+getPlayer1Pieces(): ArrayList
+getPlayer2Pieces(): ArrayList
MoveValidator
Package::moveValidator
Attributes
private static helper class
Responsabilities
«constructor» -MoveValidator()
+getInstance(): MoveValidator
+validMove(Move: move, Board: board, boolean: whiteMove): boolean
+pieceValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+knightValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+pawnValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+rookValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+bishopValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+queenValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+sameTeam(int: x1, int: y1, int: x2, int: y2, Board: board): boolean
+collision(int: x1, int: y1, int: x2, int: y2, Board: board): boolean
+otherTeam(int: x1, int: y1, int: x2, int: y2, Board: board): boolean
+pawnCapture(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+checkKnightValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+stalemate(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+isKingCheckmated(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+checkkingValidator(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+isKingChecked(int: x1, int: y1, int: x2, int: y2, Board: board, boolean: whiteMove): boolean
+hitByRook(int: x1, int: y1, int: col, int: row, Piece: king, int: kingCol, int: kingRow, int: colVal, int: r
+hitByBishop(int: x1, int: y1, int: col, int: row, Piece: king, int: kingCol, int: kingRow, int: colVal, int:
+hitByKnight(int: col, int: row, Piece: king, int: kingCol, int: kingRow, Board board): boolean
+checkKnight(Piece: piece, Piece: king, int: col, int: row): boolean
+hitByKing(Piece: king, int: kingCol, int: kingRow, Board: board): boolean
+checkKing(Piece: piece, Piece: king): boolean
+hitByPawn(int: col, int: row, Piece: king, int: kingCol, int: kingRow, Board: board): boolean
+checkPawn(Piece: pawn, Piece: king, int: col, int: row):boolean
Player
Package::player
Attributes
#pieces: ArrayList
#numCaptured: int
#name: String
#numPieces: int
Responsabilities
«constructor» +Player()
+getPieces(): ArrayList
+getNumCaptured(): int
+getNumPieces(): int
+getName(): String
+makeMove(int: x1, int: y1, int: x2, int: y2): Move
+setNumCaptured(int: numCaptured): void
+setNumPieces(int: numPieces): void
+setPieces(ArrayList: pieces): void
+initialPieces(): ArrayList
BlackPlayer
Package::player
Attributes
#pieces: ArrayList
#numCaptured: int
#name: String
#numPieces: int
Responsabilities
«constructor» +BlackPlayer(String: name)
+getPieces(): ArrayList
+getNumCaptured(): int
+getNumPieces(): int
+getName(): String
+makeMove(int: x1, int: y1, int: x2, int: y2): Move
+setNumCaptured(int: numCaptured): void
+setNumPieces(int: numPieces): void
+setPieces(ArrayList: pieces): void
+initialPieces(): ArrayList
WhitePlayer
Package::player
Attributes
#pieces: ArrayList
#numCaptured: int
#name: String
#numPieces: int
Responsabilities
«constructor» +WhitePlayer(String: name)
+getPieces(): ArrayList
+getNumCaptured(): int
+getNumPieces(): int
+getName(): String
+makeMove(int: x1, int: y1, int: x2, int: y2): Move
+setNumCaptured(int: numCaptured): void
+setNumPieces(int: numPieces): void
+setPieces(ArrayList: pieces): void
+initialPieces(): ArrayList
Queen
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +Queen(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
Piece
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
Rook
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +Rook(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
Knight
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +Knight(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
King
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +King(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
Pawn
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +Pawn(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void
Bishop
Package::pieces
Attributes
#name: String
#column: int
#row: int
#isBlack: boolean
#pic: Image
Responsabilities
«constructor» +Bishop(int: column, int: row, boolean: isBlack)
+getPic(): Image
+getColumn(): int
+getRow(): int
+isBlack(): boolean
+getPic(): Image
+getName(): String
+setColumn(int: column): void
+setRow(int: row): void

Figure 5. UML diagram of system

2.6.2 Class dependencies

The Board class has a composition relationship with the GameThread class and the MoveValidator class. This is due to the two classes not being able to exist without containing a Board object. A game is not able to exist without a board and moves cannot be checked if there is not a board. The Board class also has an aggregation relationship with the Piece class due to the association between the two. However, this dependency is not very high due to a Board still being able to exist despite there not being any pieces present. The Board class also has direct association with the UserInput class due to them constantly needing to communicate. The abstract Piece class has implementations by each of the piece specific classes. The abstract Player class has implementations by the

BlackPlayer and WhitePlayer classes. Furthermore, the class shares a composition relationship with the GameThread class due to a game not being possible without the players. The Player class also shares an aggregation relationship with the Piece class due to a player still being able to exist without any pieces. The Move class has a composition relationship with the MoveValidator class due to it not being able to exist without a Move object. The dependencies including the GameThread and MoveValidator classes have already been mentioned.

2.6.3 Packages

I made use of a packaging system in my project in order to split classes up into sections in which they provide functionality for. An example from my project is the gui package in which the Board class and UserInput class stay. This is because these are the only two classes in the project which directly deal with the graphical user interface and both of these classes need to constantly be in communication with each other. Another example from my project is the pieces package. This package contains all of the classes which directly represent the different type of pieces in the game. For this reason, the following classes are placed in this package: Bishop, King, Knight, Pawn, Piece, Queen, Rook.

Chapter 3: **Final system**

In this section I will first briefly describe the different game mechanics used and provide a description of my user interface. I will then go into detail about each class in my system and what each class does. Game mechanics will be thoroughly described in their appropriate class subsections.

3.1 How to run the system

1. Once you have downloaded the zipped file of my project you will need to extract the PROJECT-main folder into a directory of your choosing.

2. Open up Eclipse. The version of Eclipse I have used for development was "Version 2021-12 (4.22.0)".

3. Click 'File' which is located in the top left corner and then hover over 'New' and click 'Java Project'. The project name can be changed to whatever you desire but in the JRE section, tick the "Use an execution environment JRE:" and select 'JavaSE-1.7'.

4. Click the 'Next' button and click on the 'Libraries' tab. Click the 'Add Library' button and select 'JUnit' and use 'JUnit 4'. You can then click the 'Finish' button and this will create a project directory. You should be able to see this in the 'Project Explorer' section on the left. Note: This step will require the installation of JUnit in your version of Eclipse.

5. Hover over the name of the project, right click and click 'File System'. Hit the 'Next' button and in the 'From directory:' section, click the 'Browse' button and find the 'PROJECT-main' folder in your system. Make sure the option "Overwrite existing resources without warning" is ticked.

6. The project files should now be in the project directory. Now expand the src folder and expand the main package. The 'Driver.java' class should be seen.

7. Right click on 'Driver.java' and hover over the 'Run As' section. Then click on the 'Java Application' button. This should now run the Driver class and therefore display my system to you.

These are the steps used for setting up the Eclipse workspace and running the software. A full walkthrough of the software is available via the link in the 'README.mb' file in the GitLab repository and the same link is also available underneath the table of contents.

3.2 Movement mechanics

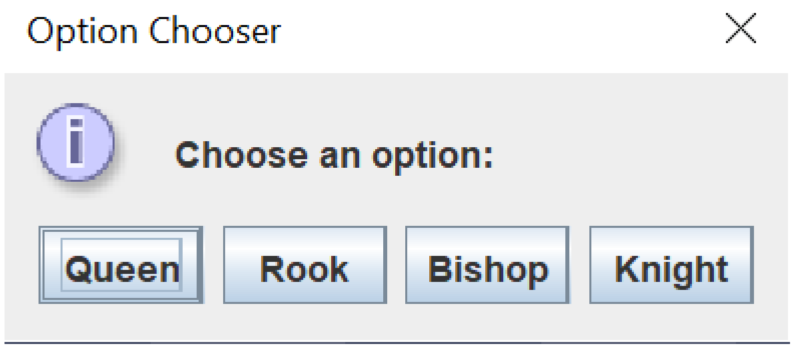
This subsection focuses on normal movement mechanics without outside interference from things such as checks or captures(these will be described later). In my system, each piece on the board is able to move exactly how they should according to the rules of chess. Pawns are only able to move forward(relative to the team they are on). The first time a pawn moves in a game, it is able to move up to 2 tiles forward. If a pawn is blocked by a piece which is directly ahead of it, regardless of the colour, the pawn will be unable to move until that tile becomes free. Knights are able to move both forwards and backwards and their movement resembles the shape of an L. This could mean a knight is able to move two tiles forwards/backwards and one tile left/right or two tiles left/right and one tile forwards/backwards. Knights do not need to worry about collisions unless the designated destination tile contains a piece. Rooks are able to move either forwards, backwards, left or right but cannot do this in a diagonal movement. Rooks cannot collide with other pieces in the process of reaching a designated destination tile. Bishops are able to move diagonally in any direction but cannot collide with any other piece in the process of reaching a destination tile. Queens combine the movement mechanics of bishops and rooks to allow it to move either diagonally in any direction or non-diagonally in any direction. Queens also are not able to collide with other pieces while on their way to a different destination tile. Kings can move one tile either forwards, backwards, left, right or diagonally in any direction. Due to collisions not being possible in the movement of one tile, Kings do not need to worry about this but they are not able to land on a destination tile that contains a teammate piece.

3.3 Capturing mechanics

Pawns are able to capture pieces exactly one tile diagonally to the right or left of them. When capturing an opponent piece, the piece is removed from the board and the piece that did the capturing is able to land on the tile previously held by the captured piece. Knights are able to capture pieces if they reside on a possible destination tile based on its movement mechanics. Rooks are able to capture pieces if there are no collisions between themselves and the piece they want to catch and the movement is in accordance with its movement mechanics. Bishops and queens share the same capture logic as rooks. Kings are able to capture any piece that resides on a possible destination tile based on its movement mechanics. This therefore means a king can capture any opponent piece if it is directly adjacent to the king.

3.4 Pawn promotion

When a pawn makes its way to the complete other side of the board, upon reaching the outer tile the user is presented with a choice of piece type to promote the pawn into. The pawn is able to promote into a queen, knight, bishop or rook. Once the user has made their choice, the movement into the edge tile is successful and the pawn is changed into the type of piece chosen by the user. The following image showcases pawn promotion in my system:


Option Chooser

Choose an option:

Figure 6. Pawn promotion options

Figure 6 refers to the promotion options a pawn has when it is able to promote itself. The option chosen is what the pawn will turn into after the movement is completed.

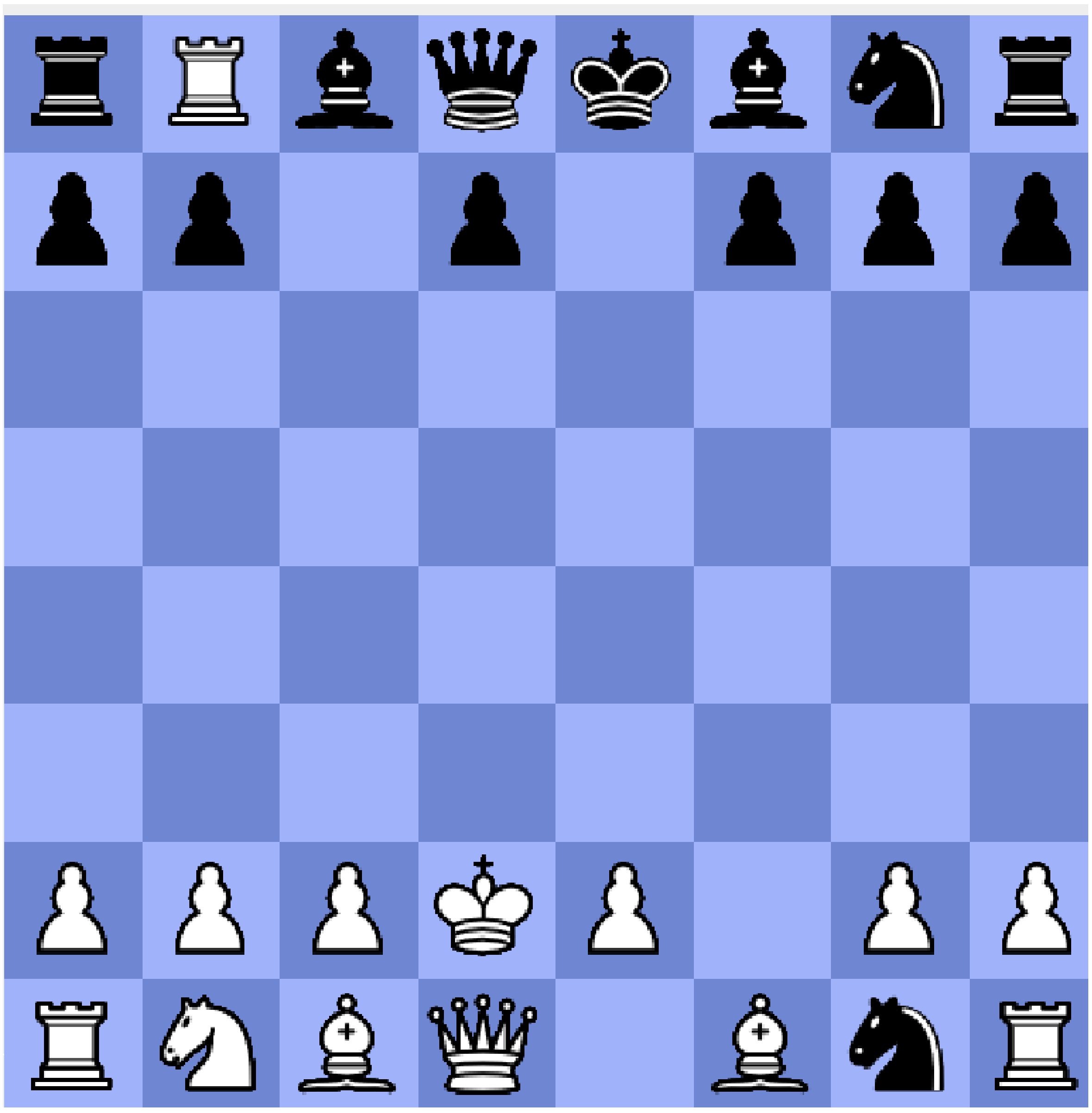


Figure 7. Pawn promotion results

Figure 7 refers to the state of the board after a promotion has taken place. In this case there have been 2 promotions. A white pawn had traversed to a tile at the very top and was able to promote itself into a rook. This is seen in tile (1,0). A black pawn has also traversed to a tile at the edge of the opposing team's side of the board. This pawn was promoted into a knight as seen in tile (6,7). Note: coordinates (x,y) refer to the x value and y value of a tile. The column on the far left represents an x value of 0 and the column on the far right represents an x value of 7. The row at the very top represents a y value of 0 and the row at the very bottom represents a y value of 7.

3.5 Check logic

When the king is currently vulnerable to a capture attack from the other team within the next round, it is in the state of check. If a player has their king in check, they are unable to perform any movements unless that movement gets the king out of check. My system also invalidates moves that would place a king into check. The following screenshot from my system showcases a check scenario:

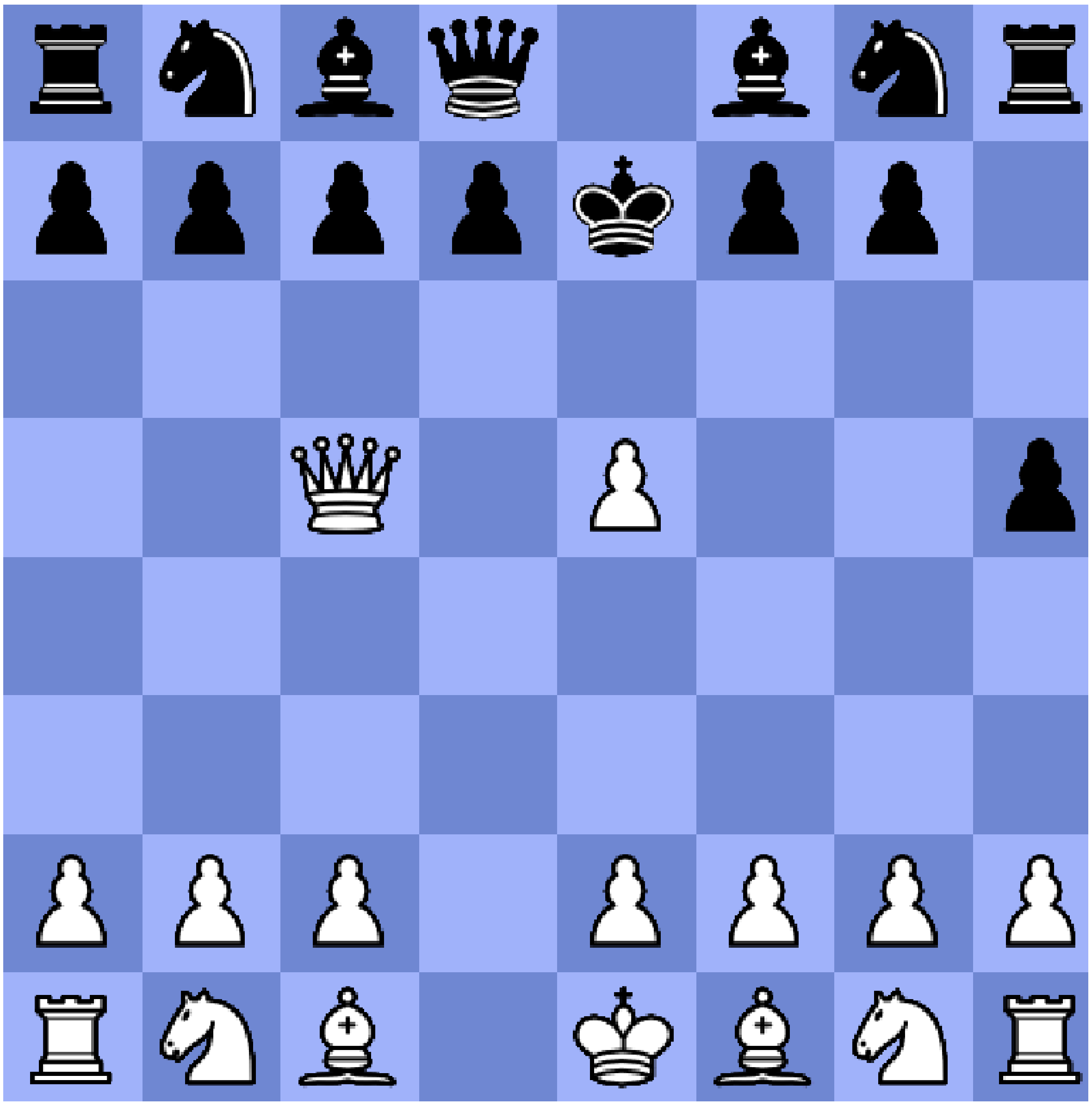


Figure 8. Check example

Figure 8 showcases the black king piece (4,1) currently in the check state. This is because the white queen (2,3) is able to capture it due to a non-collision diagonal movement being available to the king's tile. The player with the black pieces is required to make a movement that gets the king out of check. This could be moving the king into tile (4,0) or tile (4,2). The player could also move a different black piece to block the check. The only example in this case is to move black pawn (3,1) into tile (3,2) because this will block the white queen from having a legal movement onto the black king's tile.

3.6 Checkmate logic

A checkmate occurs when a player already has their king in the state of check and there are no valid moves for any of that player's pieces which takes the king out of check. The following screenshot from my project showcases a checkmate scenario:

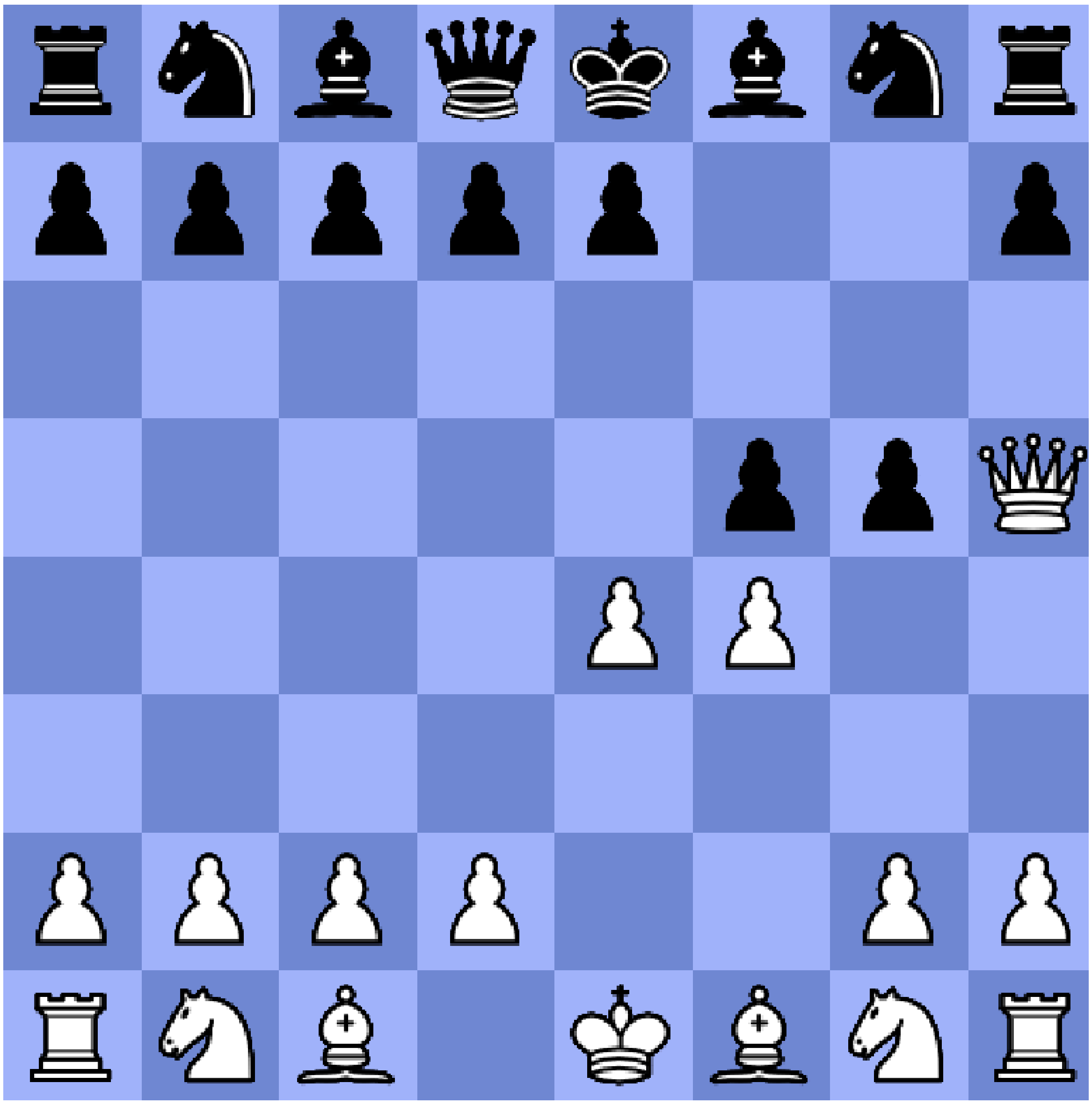


Figure 9. Checkmate example

In Figure 9, the black team is in the state of checkmate because the king is currently in check and none of the black pieces can get the king out of check. Following a checkmate, the game immediately ends and the team vulnerable to the checkmate is the loser. Therefore, in Figure 9, the white team is the winner and the black team is loser.

3.7 Stalemate logic

A stalemate occurs when the current moving team does not have any valid moves left and their respective king piece is not in the state of check. A stalemate usually occurs when any potential movement made by a player will put the king into check (which is an invalid move). The following screenshot from my project showcases a stalemate scenario:

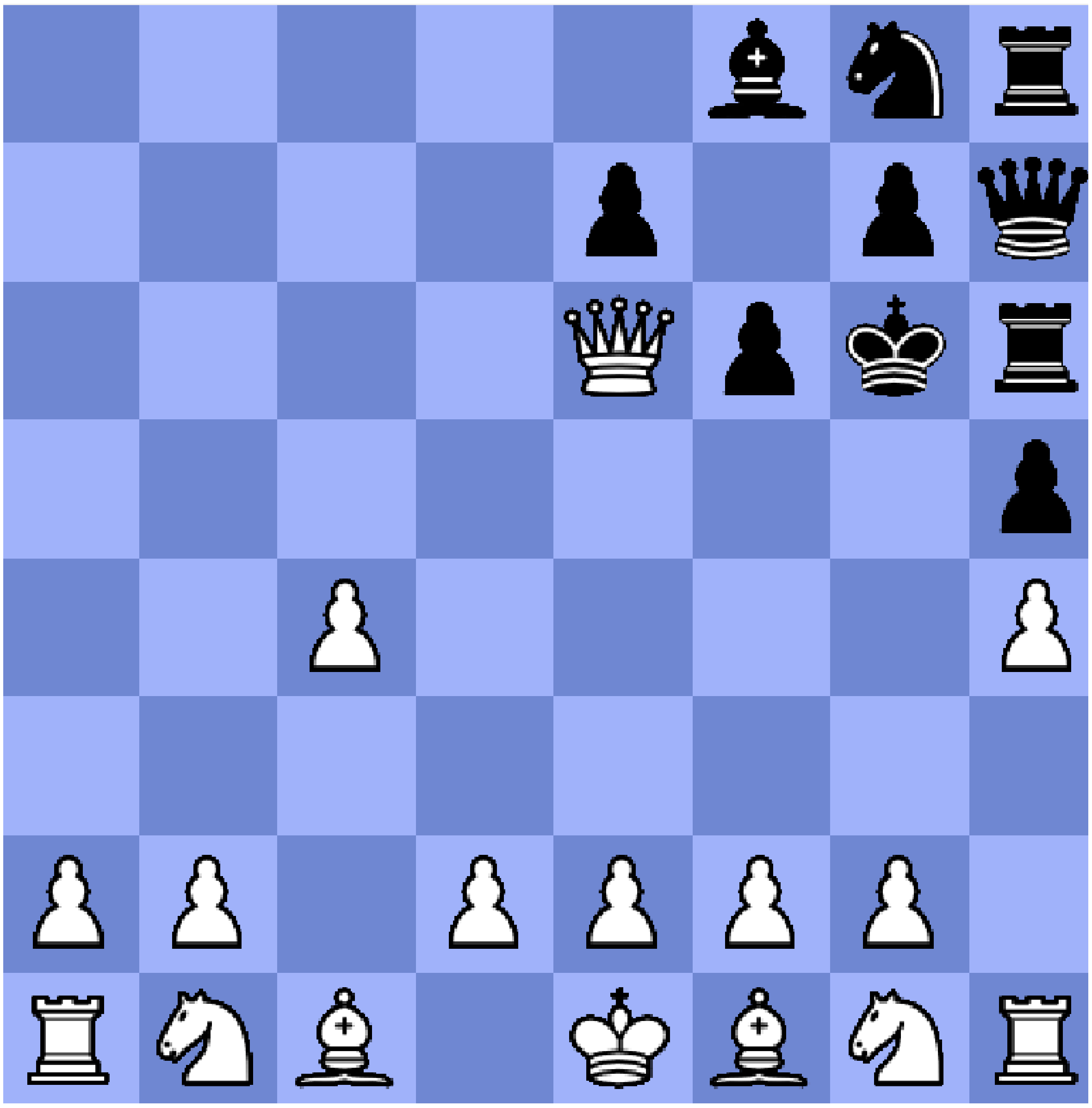


Figure 10. Stalemate example

Figure 10 showcases the black player being in a stalemate scenario. The black king is not currently in check but the player does not have any valid moves because any potentially legal moves put the king in the state of check. An example of this is moving pawn (5,2) into tile (5,3) but this is invalid due to it resulting in the white queen putting the black king into check. Another example of this is moving the black king into tile (6,3) but this move is invalid due to it resulting in the white pawn (7,4) putting the black king into check. Following a stalemate scenario, no team is the winner due to the game ending in a draw.

3.8 Game over logic

A game is over once there is an instance of either checkmate or stalemate within the game. If a checkmate occurs, the game has a winner and this winner will be displayed to the players. However, if a stalemate occurs, the game has no winner. The following examples showcase this functionality in my system:

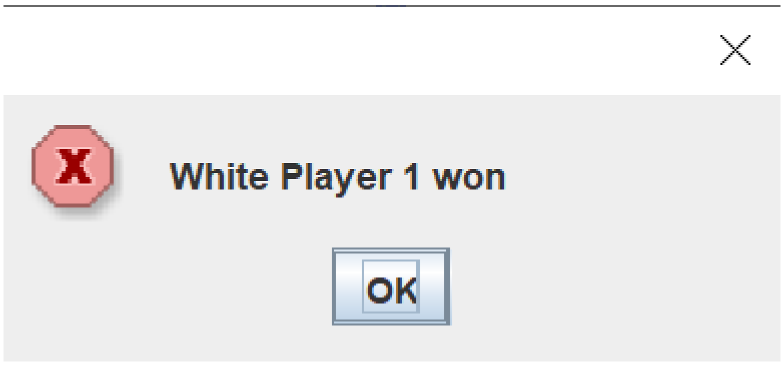
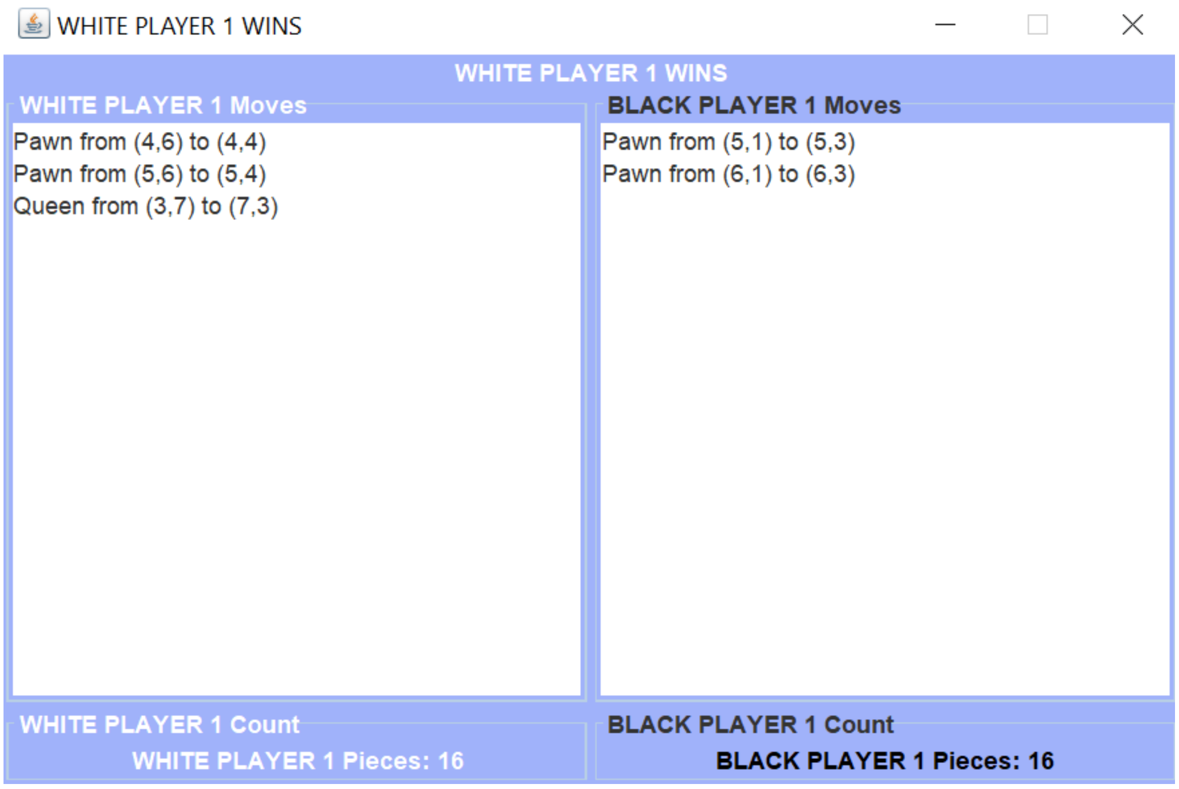

White Player 1 won

Figure 11. Game over via checkmate

Figure 11 shows that the game does have a winner following a checkmate scenario in the game. In this case, this is the result of the checkmate shown in Figure 9.


WHITE PLAYER 1 WINS

WHITE PLAYER 1 WINS

WHITE PLAYER 1 Moves

```
Pawn from (4,6) to (4,4)
Pawn from (5,6) to (5,4)
Queen from (3,7) to (7,3)
```

BLACK PLAYER 1 Moves

```
Pawn from (5,1) to (5,3)
Pawn from (6,1) to (6,3)
```

WHITE PLAYER 1 Count

WHITE PLAYER 1 Pieces: 16

BLACK PLAYER 1 Count

BLACK PLAYER 1 Pieces: 16

Figure 12. Game over via checkmate(2)

Figure 12 is a follow up from Figure 11 where the game has ended via a checkmate. This screenshot shows the winner of the game in the game status window. In this, the white player was victorious due to putting the black king in checkmate.

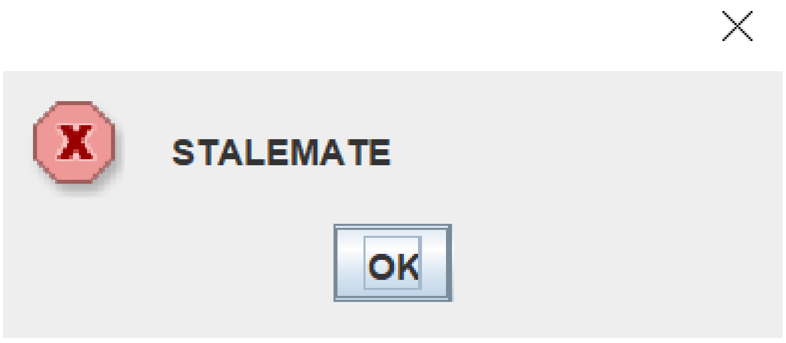
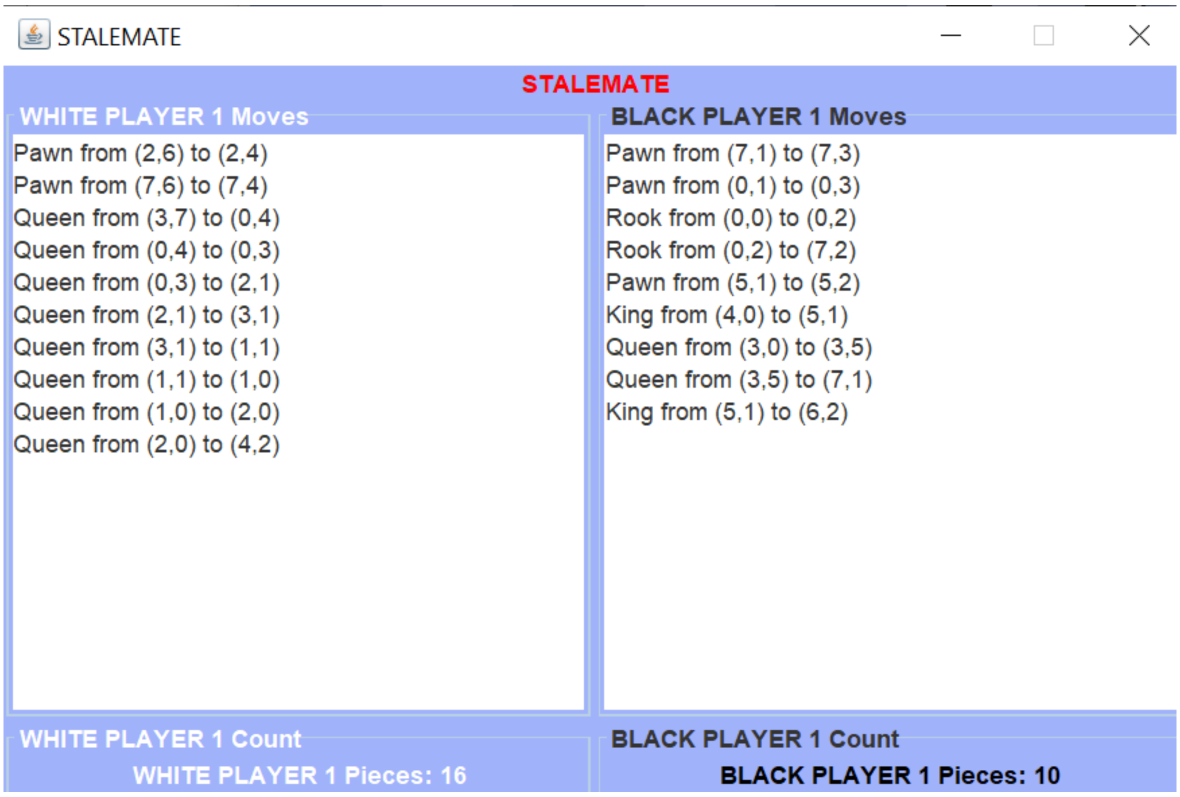


Figure 13. Game over via stalemate

Figure 13 shows that the game does not have a winner when it comes to a stalemate scenario because it just informs the users that a stalemate has occurred but nobody has won the game.



STALEMATE

WHITE PLAYER 1 Moves	BLACK PLAYER 1 Moves
Pawn from (2,6) to (2,4)	Pawn from (7,1) to (7,3)
Pawn from (7,6) to (7,4)	Pawn from (0,1) to (0,3)
Queen from (3,7) to (0,4)	Rook from (0,0) to (0,2)
Queen from (0,4) to (0,3)	Rook from (0,2) to (7,2)
Queen from (0,3) to (2,1)	Pawn from (5,1) to (5,2)
Queen from (2,1) to (3,1)	King from (4,0) to (5,1)
Queen from (3,1) to (1,1)	Queen from (3,0) to (3,5)
Queen from (1,1) to (1,0)	Queen from (3,5) to (7,1)
Queen from (1,0) to (2,0)	King from (5,1) to (6,2)
Queen from (2,0) to (4,2)	

WHITE PLAYER 1 Count	BLACK PLAYER 1 Count
WHITE PLAYER 1 Pieces: 16	BLACK PLAYER 1 Pieces: 10

Figure 14. Game over via stalemate(2)

Figure 14 is a follow up from Figure 13 in which the game has ended via a stalemate. This screenshot also shows that there is no winner following a stalemate since the title and heading both just say stalemate.

3.9 User interface

The system contains a graphical user interface which was made by making use of the swing toolkit. The chess environment can support up to 4 games of chess and therefore the interface will look slightly different depending on how many games are played. The following screenshots showcase the different interface looks depending on the number of games:

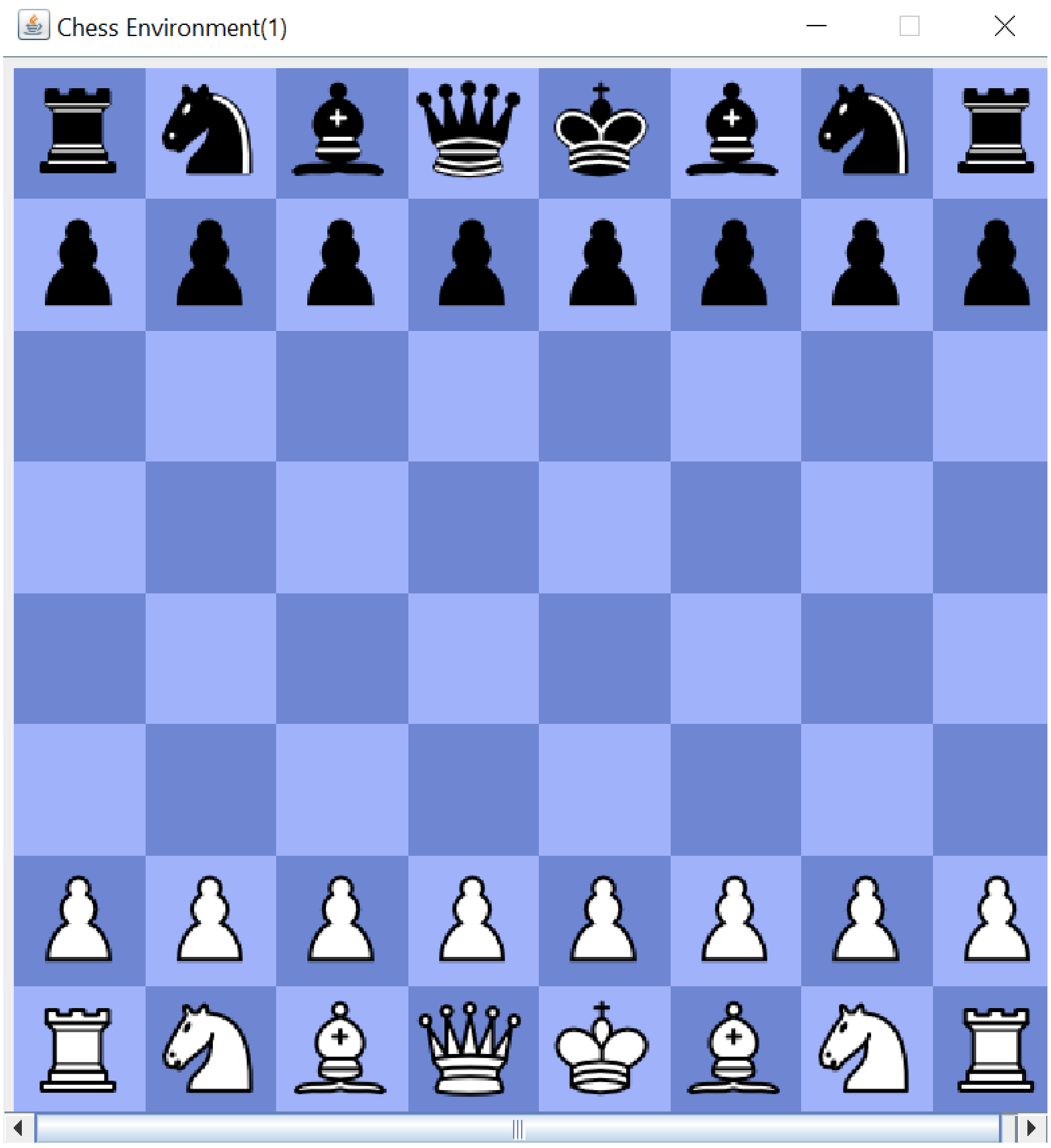


Figure 15. Singular game

Figure 15 shows the look of the graphical user interface when only one game is selected for the game environment. The number in the title represents the number of games in the environment.

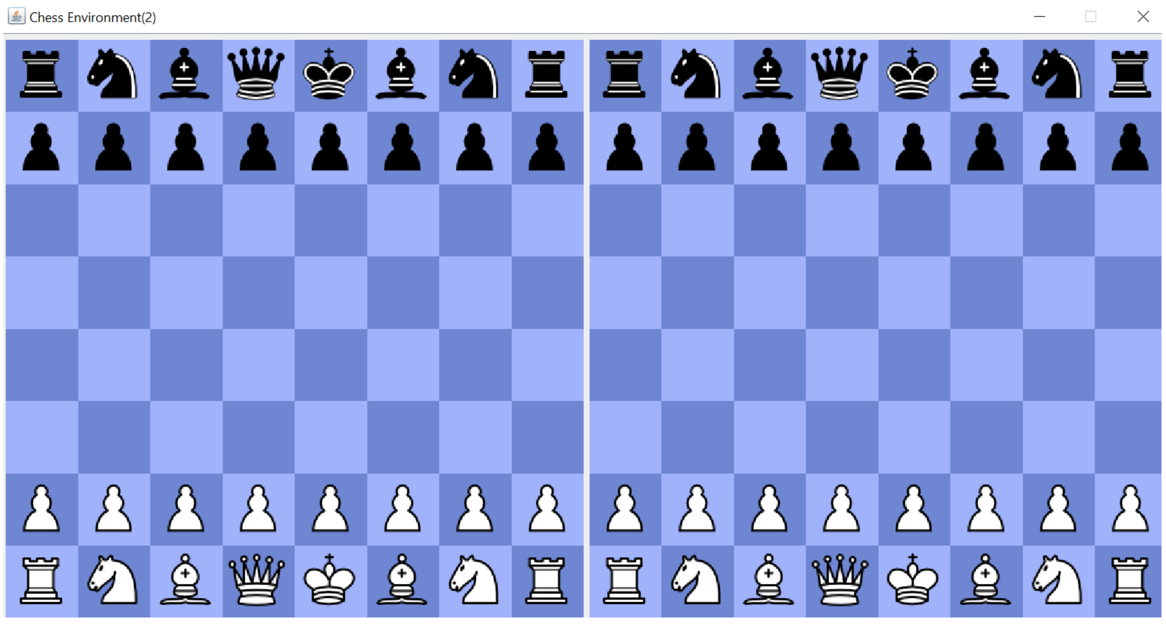


Figure 16. Two games

Figure 16 shows the look of the graphical user interface when two games of chess are created in the game environment. The number 2 in the title shows that there are two games in the environment.

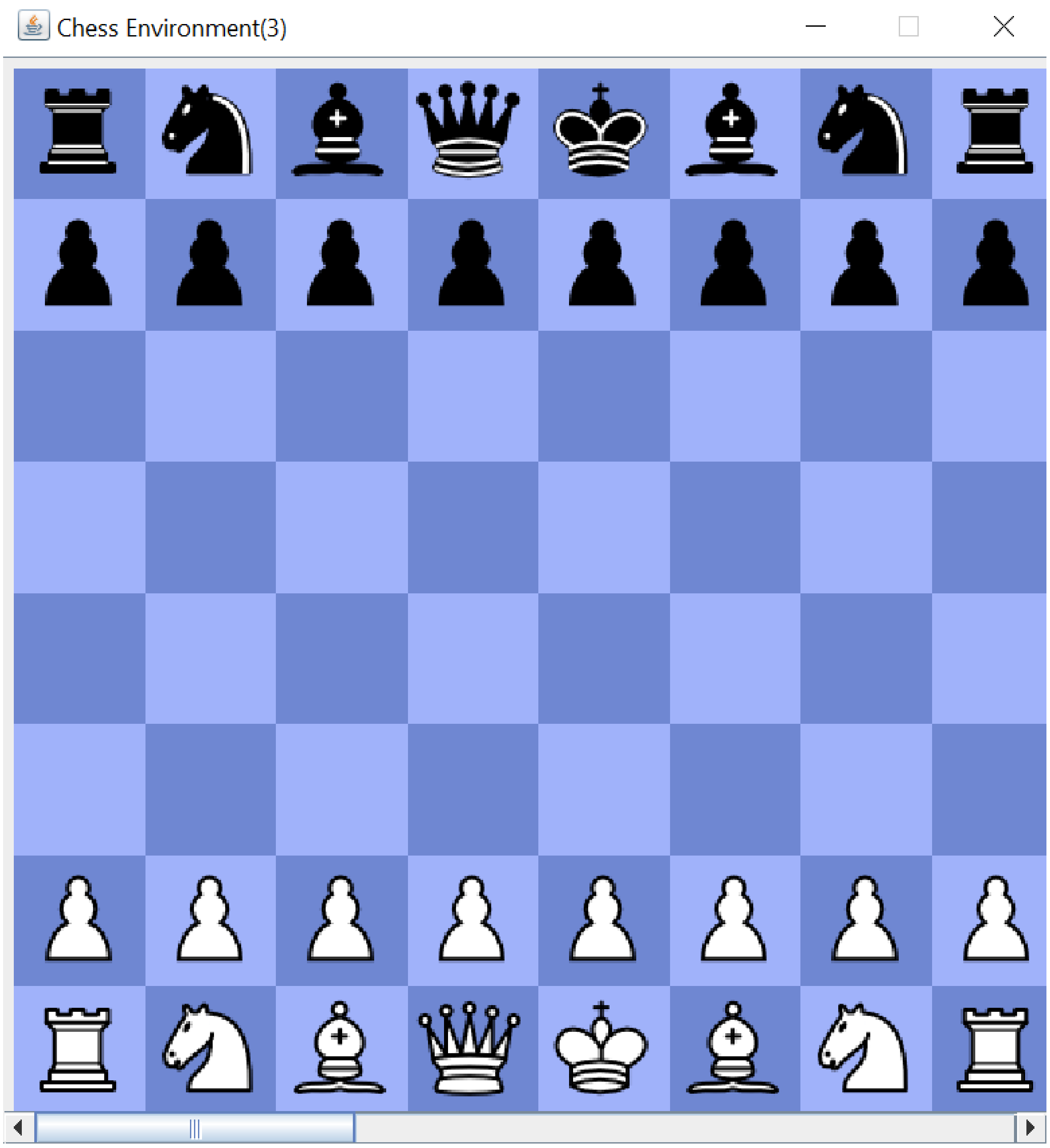


Figure 17. Three games

Figure 17 shows the look of the graphical user interface when three games of chess are chosen for the game environment. The main difference between this and Figure 15 is that this horizontal scroll bar is a lot further on the left and therefore by scrolling to the right, the other two boards will become visible. The number 3 in the title shows that there are three games in the environment.

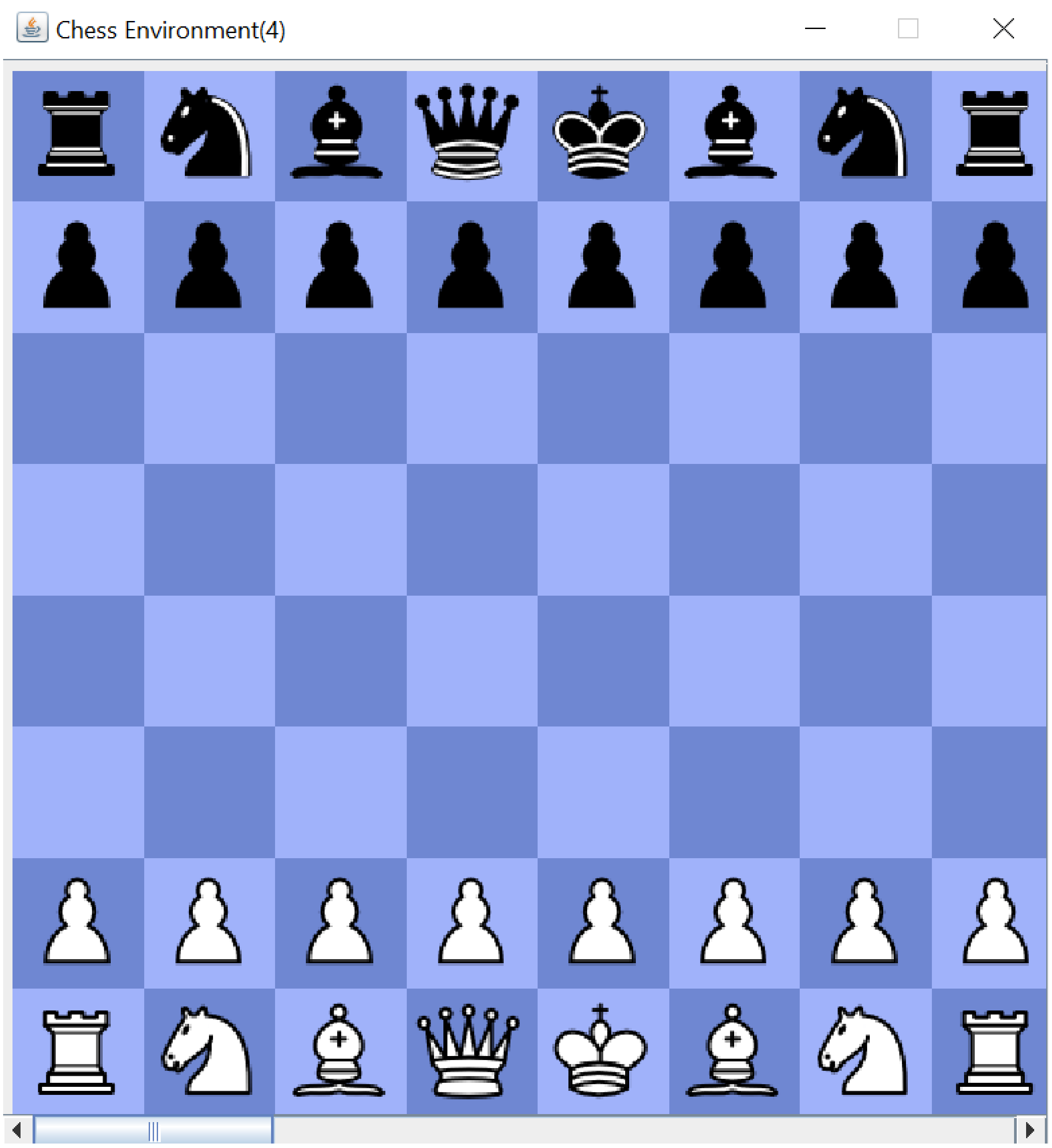
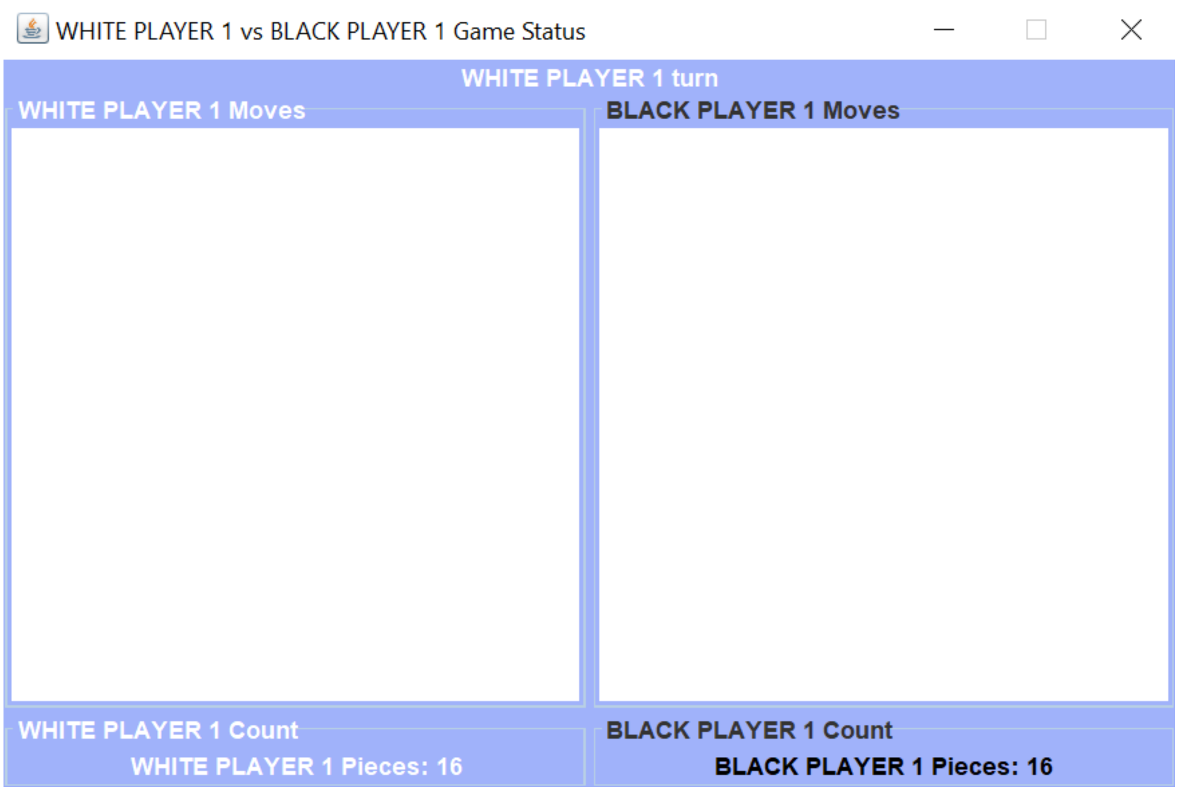


Figure 18. Four games

Figure 18 shows the look of the graphical user interface when four games of chess are chosen for the game environment. Just like Figure 17, there is a big horizontal scrollbar and in this instance, if you scroll to the right, you will be able to see all four games of chess. The number 4 in the title shows that there are four games of chess in the environment.

The interface also shows a game status window for each game in the environment. The following screenshot shows an example of a game status window in my project:


WHITE PLAYER 1 vs BLACK PLAYER 1 Game Status

WHITE PLAYER 1 turn

WHITE PLAYER 1 Moves

BLACK PLAYER 1 Moves

WHITE PLAYER 1 Count

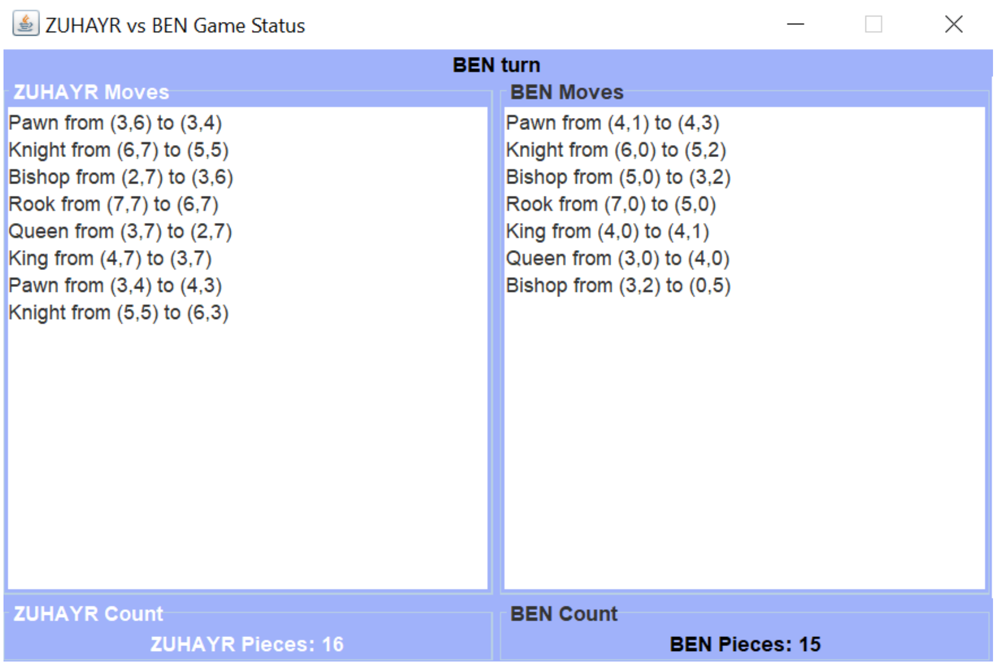
WHITE PLAYER 1 Pieces: 16

BLACK PLAYER 1 Count

BLACK PLAYER 1 Pieces: 16

Figure 19. Game status window

Figure 19 shows the look of a game status window. In this case, it is the status window for the game between "WHITE PLAYER 1" and "BLACK PLAYER 1". The Heading text is used to show who's turn it currently is. In this case it is the white player's turn. The sections "WHITE PLAYER 1 Moves" and "BLACK PLAYER 1 Moves" show the move history of each player. The bottom sections are used to show the number of pieces each player has. This example in Figure 19 is of a brand new game since there are no moves in the move history and both players still have 16 pieces. The following screenshot showcases the game status for a different game:


ZUHAYR vs BEN Game Status

BEN turn

ZUHAYR Moves

Pawn from (3,6) to (3,4)
Knight from (6,7) to (5,5)
Bishop from (2,7) to (3,6)
Rook from (7,7) to (6,7)
Queen from (3,7) to (2,7)
King from (4,7) to (3,7)
Pawn from (3,4) to (4,3)
Knight from (5,5) to (6,3)

BEN Moves

Pawn from (4,1) to (4,3)
Knight from (6,0) to (5,2)
Bishop from (5,0) to (3,2)
Rook from (7,0) to (5,0)
King from (4,0) to (4,1)
Queen from (3,0) to (4,0)
Bishop from (3,2) to (0,5)

ZUHAYR Count

ZUHAYR Pieces: 16

BEN Count

BEN Pieces: 15

Figure 20. Another game status window

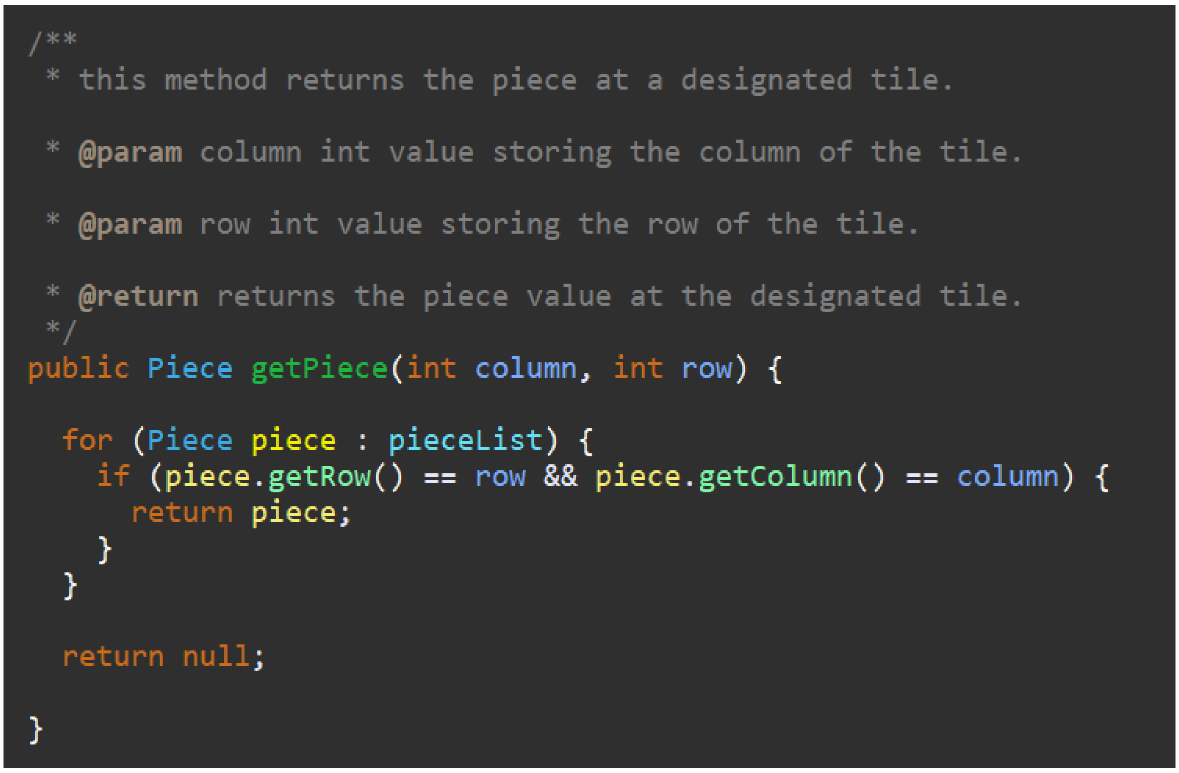
Figure 20 shows the look of a different game status window. In this example, the white and black players have each been given a name and therefore 'ZUHAYR' is the white player and 'BEN' is the black player. This example shows that it is currently Ben's turn to move and it also shows the move history for each player. This move history shows the piece that moven in a round and the source and destination tiles for the movement. This example shows that Ben has 15 pieces which means one of his pieces was successfully captured by Zuhayr.

3.10 Class descriptions

In this subsection I will dive into each individual class in my project and talk about the functionality and importance of each.

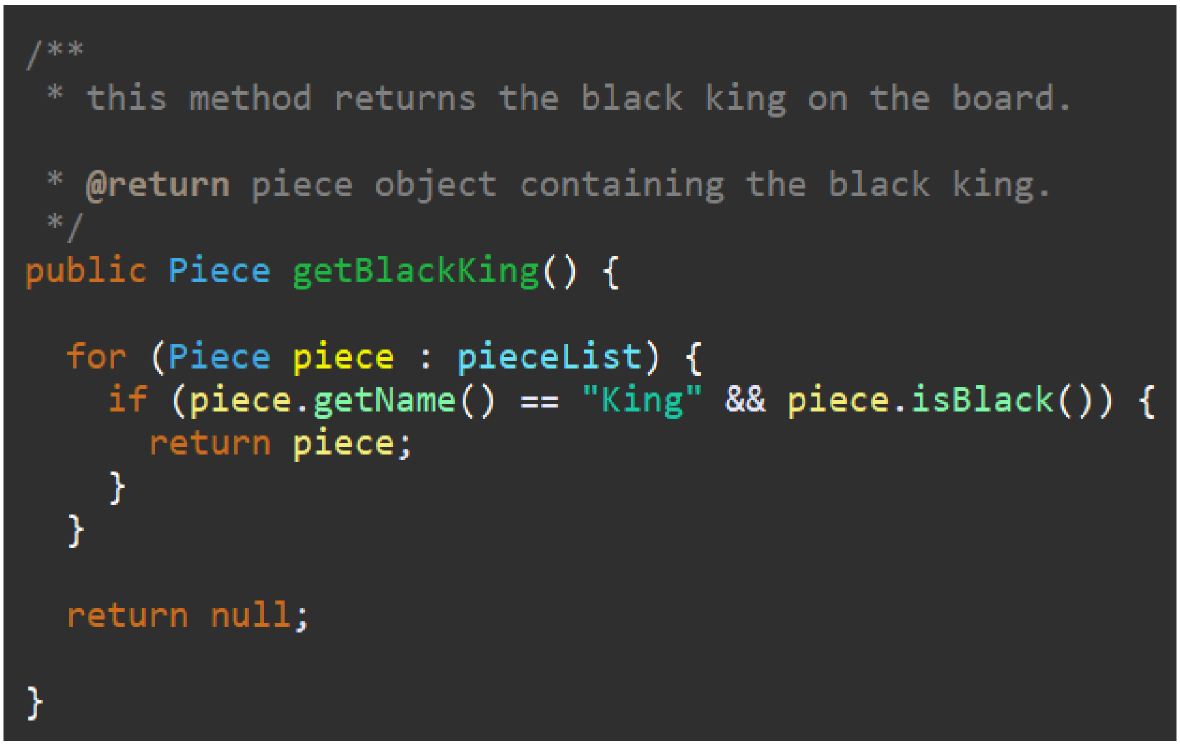
3.10.1 GUI classes

The board class belongs to the gui package and it is used to represent the graphical interface of the chess board. It makes use of the awt package in java to create the user interface and paint graphics and images onto the screen. The primary purpose of this class is to draw a chess board on the screen and it makes use of the UserInput class to manipulate user inputs on the interface such as mouse clicks. This connection between the Board class and the UserInput class is particularly useful because it allows the Board to store coordinates of selected pieces and also allows the Board to keep track of when the Board has been pressed and released. The UserInput class uses the designated getters and setters to change the attribute values of the Board class when the interface is interacted with. The board class also has methods such as getPiece or getWhiteKing which are very useful especially when checking for move legality in the move validator class. The following methods are all particularly useful for external use(used by other classes) in each game:



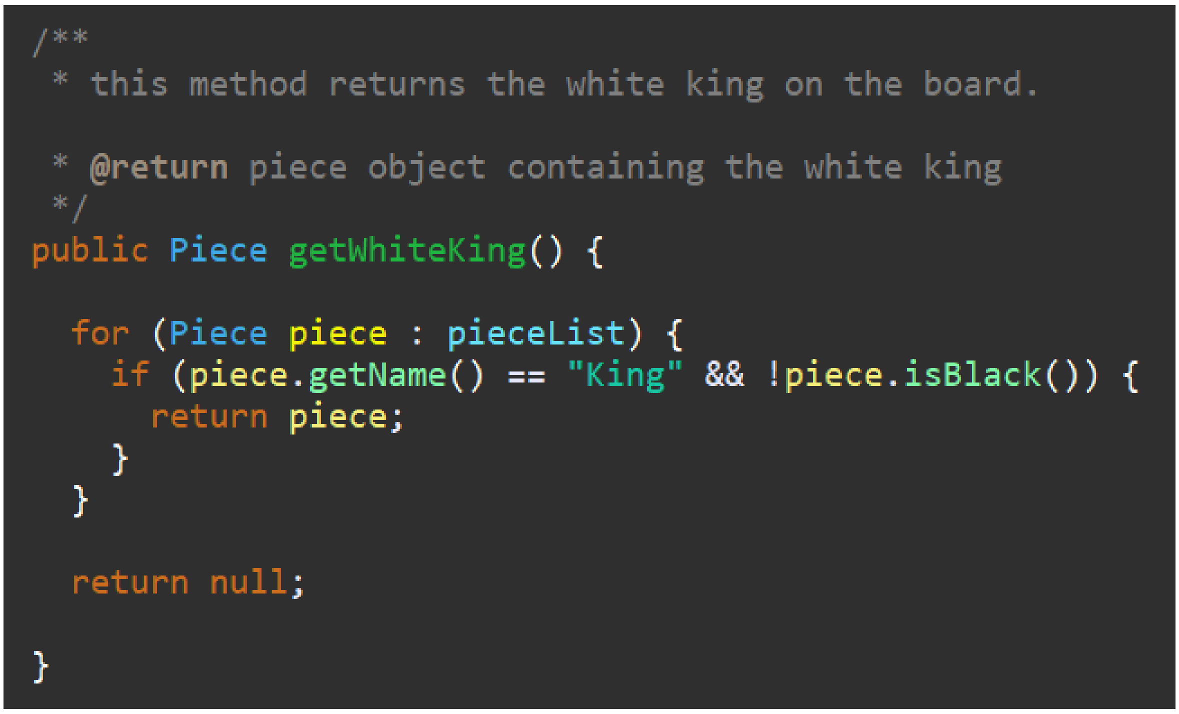
```
/**
 * this method returns the piece at a designated tile.

 * @param column int value storing the column of the tile.

 * @param row int value storing the row of the tile.

 * @return returns the piece value at the designated tile.
 */
public Piece getPiece(int column, int row) {

  for (Piece piece : pieceList) {
    if (piece.getRow() == row && piece.getColumn() == column) {
      return piece;
    }
  }

  return null;

}
```

Figure 21. Get piece method



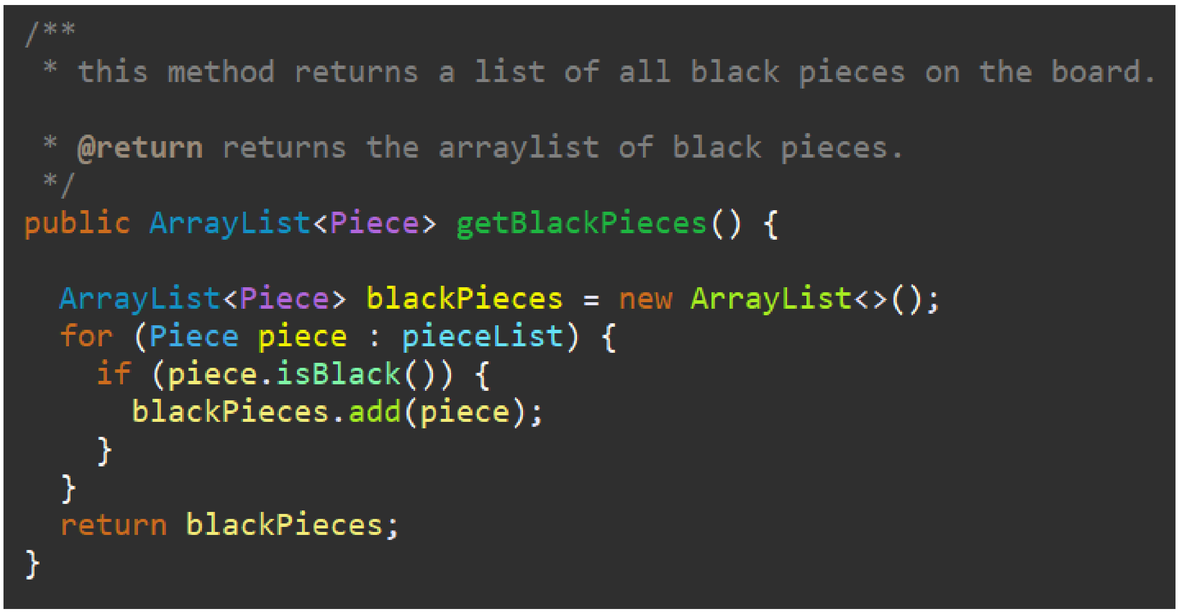
```java
/**
 * this method returns the black king on the board.

 * @return piece object containing the black king.
 */
public Piece getBlackKing() {

  for (Piece piece : pieceList) {
    if (piece.getName() == "King" && piece.isBlack()) {
      return piece;
    }
  }

  return null;

}
```

Figure 22. Get black king method



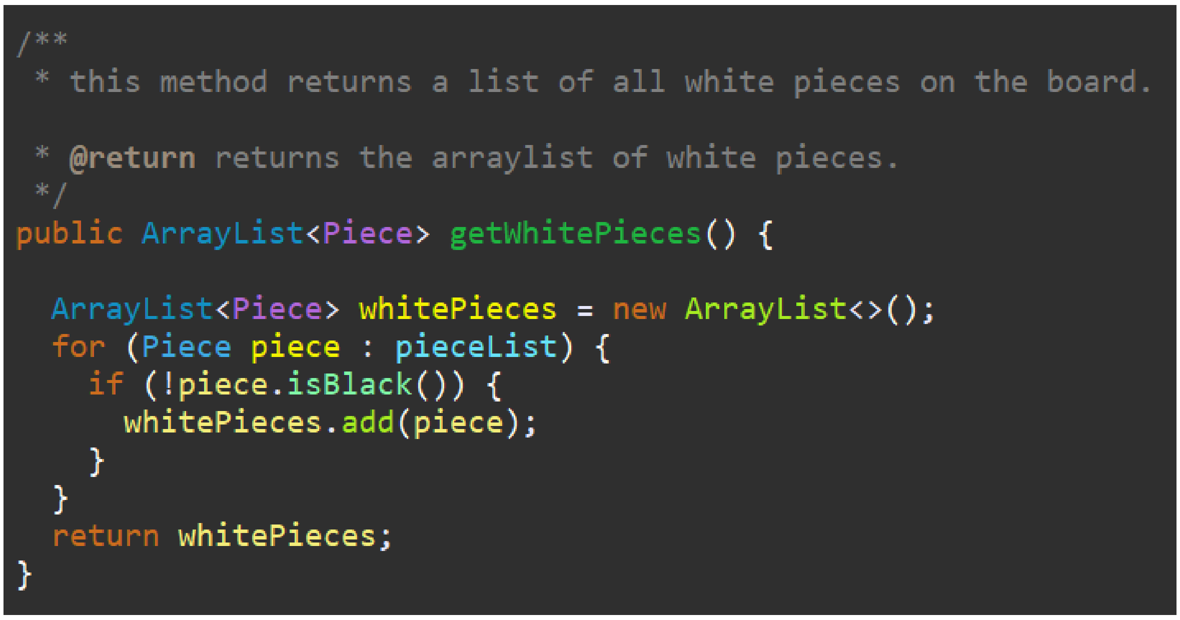
```java
/**
 * this method returns the white king on the board.

 * @return piece object containing the white king
 */
public Piece getWhiteKing() {

  for (Piece piece : pieceList) {
    if (piece.getName() == "King" && !piece.isBlack()) {
      return piece;
    }
  }

  return null;

}
```

Figure 23. Get white king method



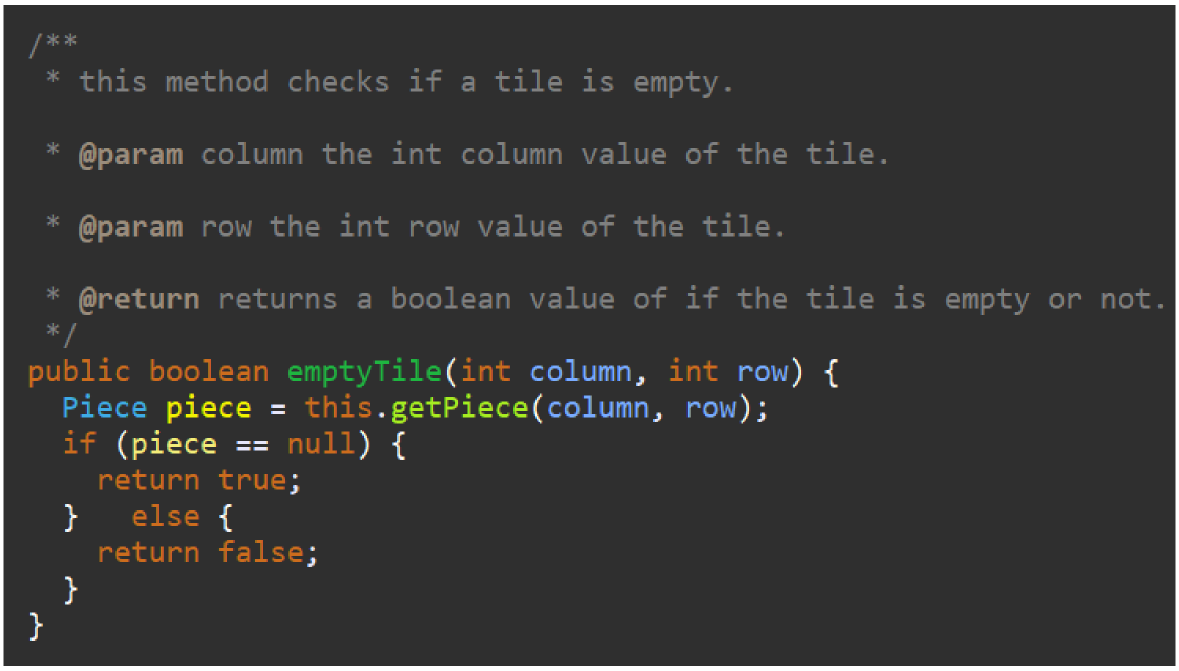
```java
/**
 * this method returns a list of all black pieces on the board.

 * @return returns the arraylist of black pieces.
 */
public ArrayList<Piece> getBlackPieces() {

  ArrayList<Piece> blackPieces = new ArrayList<>();
  for (Piece piece : pieceList) {
    if (piece.isBlack()) {
      blackPieces.add(piece);
    }
  }
  return blackPieces;
}
```

Figure 24. Get black pieces method



```java
/**
 * this method returns a list of all white pieces on the board.

 * @return returns the arraylist of white pieces.
 */
public ArrayList<Piece> getWhitePieces() {

  ArrayList<Piece> whitePieces = new ArrayList<>();
  for (Piece piece : pieceList) {
    if (!piece.isBlack()) {
      whitePieces.add(piece);
    }
  }
  return whitePieces;
}
```

Figure 25. Get white pieces method



```
/**
 * this method checks if a tile is empty.

 * @param column the int column value of the tile.

 * @param row the int row value of the tile.

 * @return returns a boolean value of if the tile is empty or not.
 */
public boolean emptyTile(int column, int row) {
  Piece piece = this.getPiece(column, row);
  if (piece == null) {
    return true;
  }   else {
    return false;
  }
}
```

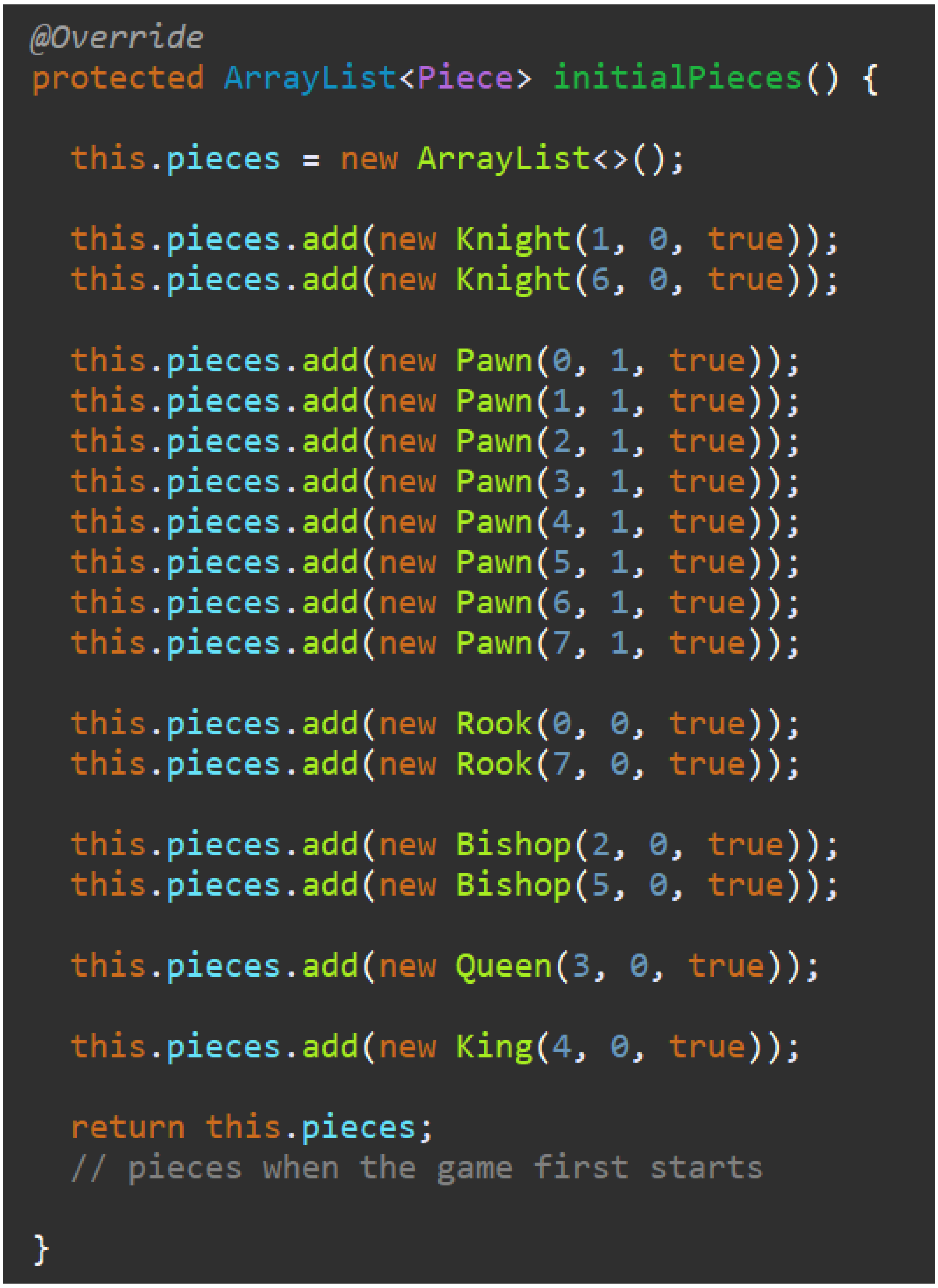
Figure 26. Empty tile method

3.10.2 Piece classes

The Piece class is an abstract class used as a blueprint for storing data required in each different piece. The class contains a name attribute which is used to distinguish between the different pieces. The column and row attributes store the location of each piece and the isBlack boolean is used to determine which team each piece belongs to. The final attribute stored by this abstract class is an image value which will contain a sprite of the piece. It was important to create this abstract class because it allows each specific piece class to follow the same structure and therefore achieves consistency. Each specific Piece class has its column, row and team colour passed into the constructor as arguments, which is a very effective approach because these are the main attributes that would differ between pieces of the same type. For example, each game of chess contains four bishop pieces(two on each team) and therefore the column, row and team attributes of these bishops will not be the same in any of them, hence the need of passing them into the constructor. The pieces package contains the abstract Piece class and also a Bishop, King, Knight, Pawn, Queen, Rook class which extends the abstract Piece class.

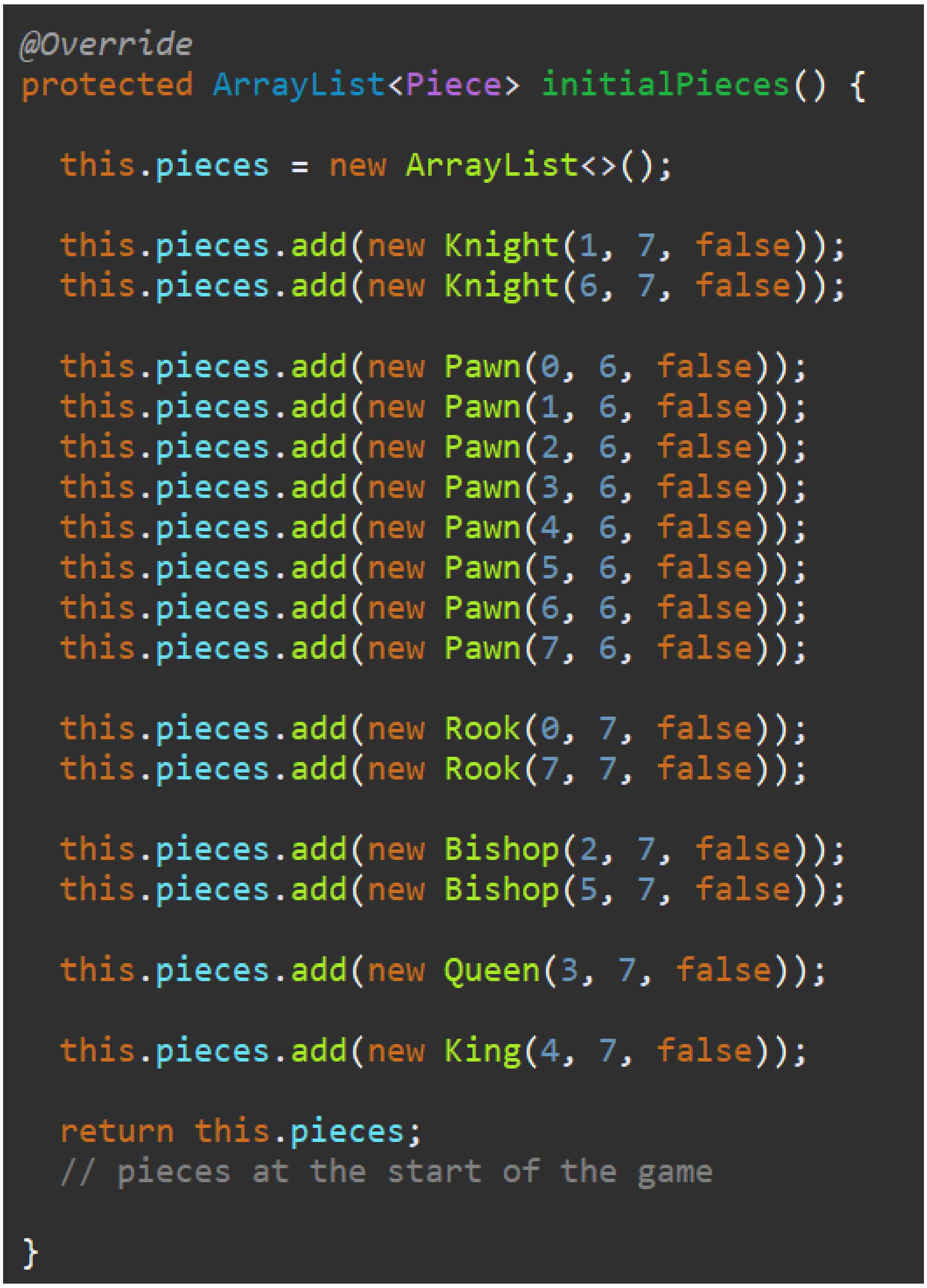
3.10.3 Player classes

The Player class is another abstract class which is used as a blueprint for storing data required in each type of player. The class contains an ArrayList of pieces which represents the pieces currently held by a particular player. The numCaptured attribute is used to store the number of opposing pieces currently captured by a particular player. This is automatically set to zero in the constructor of this abstract class due to it being impossible to start a game of chess already possessing any enemy pieces. The numPieces attribute is used to store the number of pieces currently held by a player and the name attribute stores the name of a player. There are two classes which extend this abstract Player class and they are the BlackPlayer and WhitePlayer classes. The BlackPlayer class is used to represent the player which holds the black pieces in the game. In the construction of this class, the player is able to pass in their name and the class also creates the initial pieces for the player. It does this by calling the initialPieces() function which initiates the pieces variable and adds the distinct black pieces to this ArrayList of pieces. Therefore, whenever a BlackPlayer object is created, it contains the same pieces. The WhitePlayer class serves the same purpose as the BlackPlayer class except it represents the player which holds the white pieces in the game. The getters and setters in each of these classes are very important for manipulating player data based on the outcome of each move in a game.



```
@Override
protected ArrayList<Piece> initialPieces() {

  this.pieces = new ArrayList<>();

  this.pieces.add(new Knight(1, 0, true));
  this.pieces.add(new Knight(6, 0, true));

  this.pieces.add(new Pawn(0, 1, true));
  this.pieces.add(new Pawn(1, 1, true));
  this.pieces.add(new Pawn(2, 1, true));
  this.pieces.add(new Pawn(3, 1, true));
  this.pieces.add(new Pawn(4, 1, true));
  this.pieces.add(new Pawn(5, 1, true));
  this.pieces.add(new Pawn(6, 1, true));
  this.pieces.add(new Pawn(7, 1, true));

  this.pieces.add(new Rook(0, 0, true));
  this.pieces.add(new Rook(7, 0, true));

  this.pieces.add(new Bishop(2, 0, true));
  this.pieces.add(new Bishop(5, 0, true));

  this.pieces.add(new Queen(3, 0, true));

  this.pieces.add(new King(4, 0, true));

  return this.pieces;
  // pieces when the game first starts

}
```

Figure 27. Initial pieces method for black player



```java
@Override
protected ArrayList<Piece> initialPieces() {

  this.pieces = new ArrayList<>();

  this.pieces.add(new Knight(1, 7, false));
  this.pieces.add(new Knight(6, 7, false));

  this.pieces.add(new Pawn(0, 6, false));
  this.pieces.add(new Pawn(1, 6, false));
  this.pieces.add(new Pawn(2, 6, false));
  this.pieces.add(new Pawn(3, 6, false));
  this.pieces.add(new Pawn(4, 6, false));
  this.pieces.add(new Pawn(5, 6, false));
  this.pieces.add(new Pawn(6, 6, false));
  this.pieces.add(new Pawn(7, 6, false));

  this.pieces.add(new Rook(0, 7, false));
  this.pieces.add(new Rook(7, 7, false));

  this.pieces.add(new Bishop(2, 7, false));
  this.pieces.add(new Bishop(5, 7, false));

  this.pieces.add(new Queen(3, 7, false));

  this.pieces.add(new King(4, 7, false));

  return this.pieces;
  // pieces at the start of the game

}
```

Figure 28. Initial pieces method for white player

3.10.4 Move class

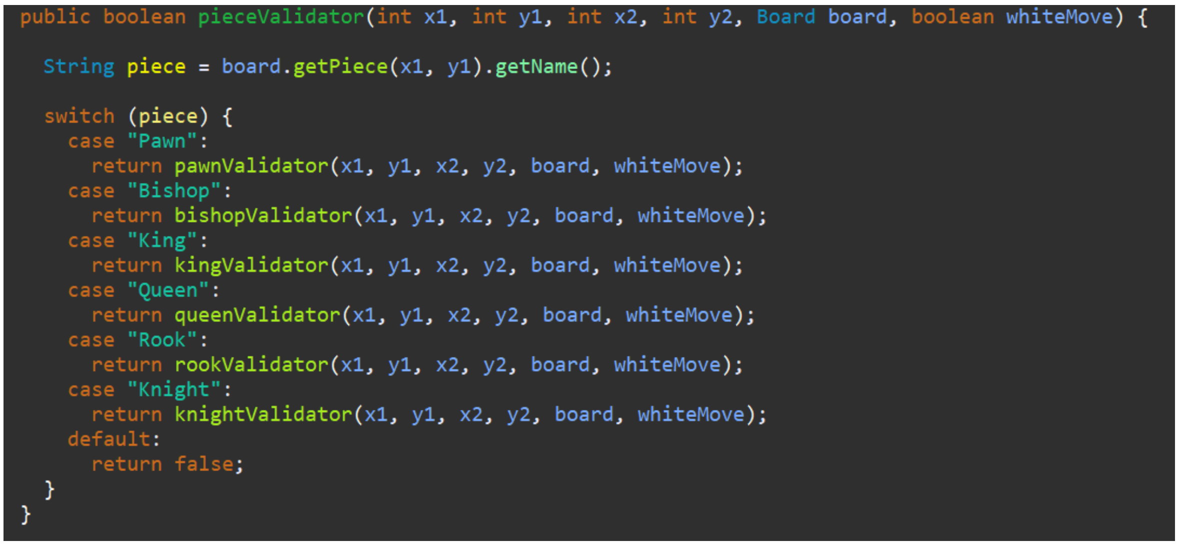
The Move class is used to store information regarding a movement in the system. It contains an x1 value which stores the initial x value of a movement and a y1 value which stores the initial y value of a movement. The class also stores an x2 and y2 value which both represent the destination x and y values of a movement. Each of these attributes are passed into the constructor.

3.10.5 MoveValidator class

The move validator class is used to validate each move attempted in a game. The class contains an empty private constructor and makes use of the Bill Pugh singleton design in order to achieve thread-safety in the creation of the class. Thread-safety is imperative for this class due to each thread requiring a reference to it in order to check if a particular move in a game is legal.

The validMove method in the class is used to return a boolean value stating whether or not a particular move is legal. This method is given a Move object, a Board object and a boolean value which distinguishes what player has made this move. This method immediately returns false if the player who is trying to move is in check or the move will result in them being in check. If this is not the case however, the method will perform other move validation. If the coordinates of the movement are out of range, this method will return a false value. Furthermore, if the board does not contain a piece in the initial coordinate of the move, or the piece in that coordinate belongs to the other team, the method will also return a false value. However, if none of these conditions are met, the coordinate values, board object and boolean value are passed into the piece validator method.

The piece validator method also returns a boolean but it is used to distinguish the type of piece being used based on the x1 and y1 values. The method makes use of the getPiece method in the Board class and calls the appropriate validator function based on the type of piece. For example, if the piece in the x1 y1 coordinate on the board is a pawn, the method will call the pawnValidator method. There are validator methods for each type of piece and they are used to determine if the movement follows the mechanics required for the type of piece. The following screenshot shows the piece validator method.



```java
public boolean pieceValidator(int x1, int y1, int x2, int y2, Board board, boolean whiteMove) {

  String piece = board.getPiece(x1, y1).getName();

  switch (piece) {
    case "Pawn":
      return pawnValidator(x1, y1, x2, y2, board, whiteMove);
    case "Bishop":
      return bishopValidator(x1, y1, x2, y2, board, whiteMove);
    case "King":
      return kingValidator(x1, y1, x2, y2, board, whiteMove);
    case "Queen":
      return queenValidator(x1, y1, x2, y2, board, whiteMove);
    case "Rook":
      return rookValidator(x1, y1, x2, y2, board, whiteMove);
    case "Knight":
      return knightValidator(x1, y1, x2, y2, board, whiteMove);
    default:
      return false;
  }
}
```

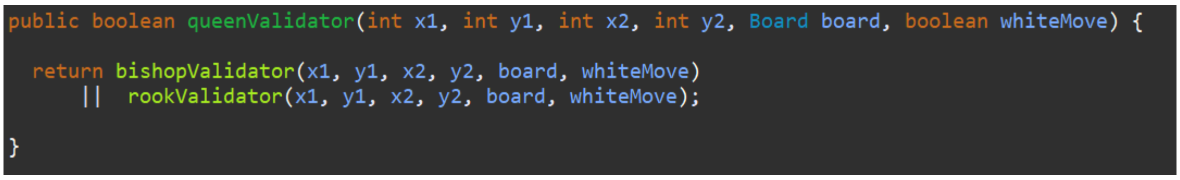
Figure 29. Piece validator method

The knight validator method returns a true value if the knight does not have destination coordinates which are shared with another piece of the same team and if the movement follows the particular mechanics required for a knight. As mentioned in a previous section, a knight is able to move in an L shape, this is logically implemented in the following way: Math.abs(x2 - x1) * Math.abs(y2 - y1) == 2. This algorithm ensures only one of the x/y values move two tile lengths whereas the other value moves exactly one tile length. This is due to the only possible values resulting in a product of two being two and one.

The rook validator method returns a true value if the rook moves either vertically or horizontally while also not colliding with any other pieces along the way and not landing on a tile occupied by a teammate piece. The method is able to determine if the movement is purely horizontal by seeing a change in the x1 x2 variables but not a change in the y1 y2 variables. Similarly, the method can determine if a move is purely vertical based on a change in the y1 y2 variables but not a change in the x1 x2 variables.

The bishop validator method returns a true value if the bishop moves perfectly diagonally while also not colliding with any other pieces and not landing on a tile occupied by a teammate piece. The following algorithm is used to determine if a bishop has moved a perfect diagonal: Math.abs(x2 - x1) == Math.abs(y2 - y1). A perfect diagonal is only achieved if the distance moved horizontally matches the distance moved vertically. For example, if a bishop was to move from the (2,2) tile to the (3,3) tile, the distance travelled horizontally would be one and the distance travelled vertically would be one. Therefore, the bishop will have moved a perfect diagonal. This method also returns a false value if the destination coordinates are identical to the source coordinates.

The queen validator method returns a true value if the movement satisfies that of a bishop or a rook. The following screenshot shows how it accomplishes this:



```
public boolean queenValidator(int x1, int y1, int x2, int y2, Board board, boolean whiteMove) {

  return bishopValidator(x1, y1, x2, y2, board, whiteMove)
      ||  rookValidator(x1, y1, x2, y2, board, whiteMove);

}
```

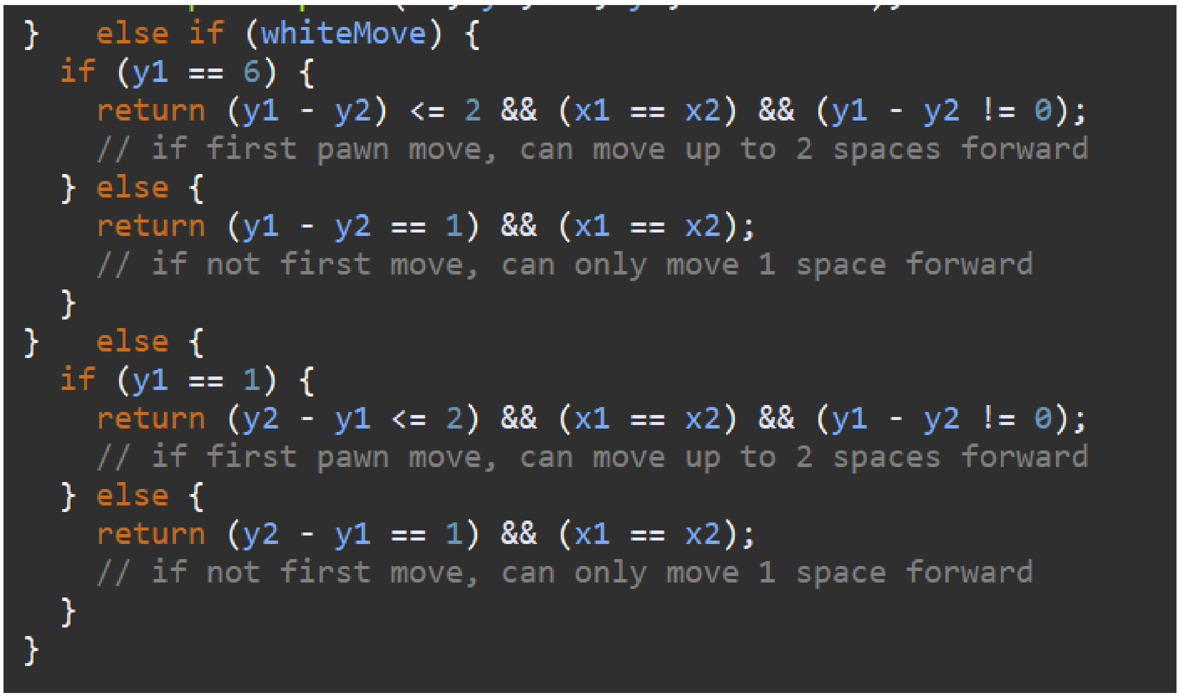
Figure 30. Queen validator method

The king validator method returns a true value if the king moves to any adjacent tile which is not occupied by a teammate piece. In order to determine if the movement is to an adjacent tile, the following algorithm is used:

```
Math.abs((x2 - x1) * (y2 - y1)) == 1 || Math.abs(x2 - x1) + Math.abs(y2 -
y1) == 1;
```

The first case that would satisfy the algorithm would be if the king moves a perfect diagonal by exactly one tile. An example that would satisfy this first case is if a king tried to move from (4,4) to (5,5). The change in x is exactly one and the change in y is exactly one. The product of these is equal to one and therefore satisfies this case. The other case that would satisfy the algorithm would be if the king moves either horizontally or vertically exactly one tile. An example that would satisfy this case is it a king tried to move from (4,4) to (4,5). This movement is exactly one tile vertically. The king validator method also returns false if the destination coordinates are identical to the source coordinates.

The pawn validator method returns a true value if a pawn follows its specific movement mechanics while also not colliding with any other pieces, landing on a tile occupied by a teammate piece or trying to forward capture an opponent piece. The following screenshot shows my algorithm for determining if the movement follows the nature of a pawn(other cases like collisions are checked prior to this):

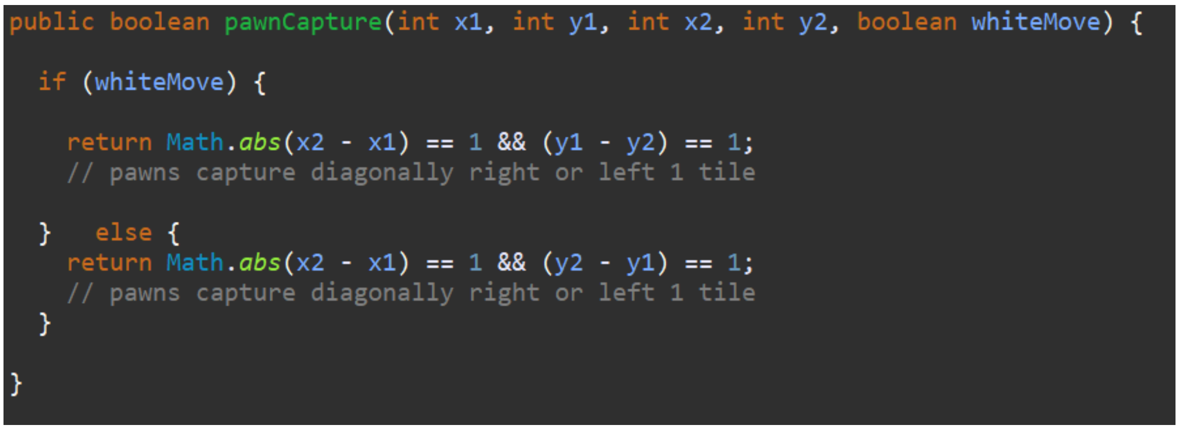


```
}   else if (whiteMove) {
  if (y1 == 6) {
    return (y1 - y2) <= 2 && (x1 == x2) && (y1 - y2 != 0);
    // if first pawn move, can move up to 2 spaces forward
  } else {
    return (y1 - y2 == 1) && (x1 == x2);
    // if not first move, can only move 1 space forward
  }
}   else {
  if (y1 == 1) {
    return (y2 - y1 <= 2) && (x1 == x2) && (y1 - y2 != 0);
    // if first pawn move, can move up to 2 spaces forward
  } else {
    return (y2 - y1 == 1) && (x1 == x2);
    // if not first move, can only move 1 space forward
  }
}
```

Figure 31. Pawn movement logic

The algorithm makes use of the boolean value passed into the method to manipulate this logic based on the player currently trying to move. If the pawn belongs to the white player and sits on the sixth column of the board it implies it has not yet been moved at all in the game. Therefore, the pawn will be able to move up to two spaces forwards. If the white pawn is on any other column, it will only be able to move one space forward(disregarding capturing). Similarly, if a black pawn sits in the first column, it will be able to move up to two tiles forwards whereas any other column would only be able to move one tile forward. If the pawn attempts to make a capture and the movement is not purely vertical, this will run the pawn capture method.

The pawn capture method returns a true value if the pawn moves a perfect diagonal exactly one tile away. This movement will be allowed because, as previously mentioned, this method is only called when the pawn is attempting a capture and the only way a pawn can capture a piece is if it is exactly one tile ahead of it diagonally. The method is shown below:



```
public boolean pawnCapture(int x1, int y1, int x2, int y2, boolean whiteMove) {

  if (whiteMove) {

    return Math.abs(x2 - x1) == 1 && (y1 - y2) == 1;
    // pawns capture diagonally right or left 1 tile

  }   else {
    return Math.abs(x2 - x1) == 1 && (y2 - y1) == 1;
    // pawns capture diagonally right or left 1 tile
  }

}
```

Figure 32. Pawn capture method

As previously mentioned, each validator method checks if a movement is destined for a tile occupied by a teammate piece and/or if the movement contains any collisions. The first case is checked using a sameTeam method which makes use of the following algorithm:

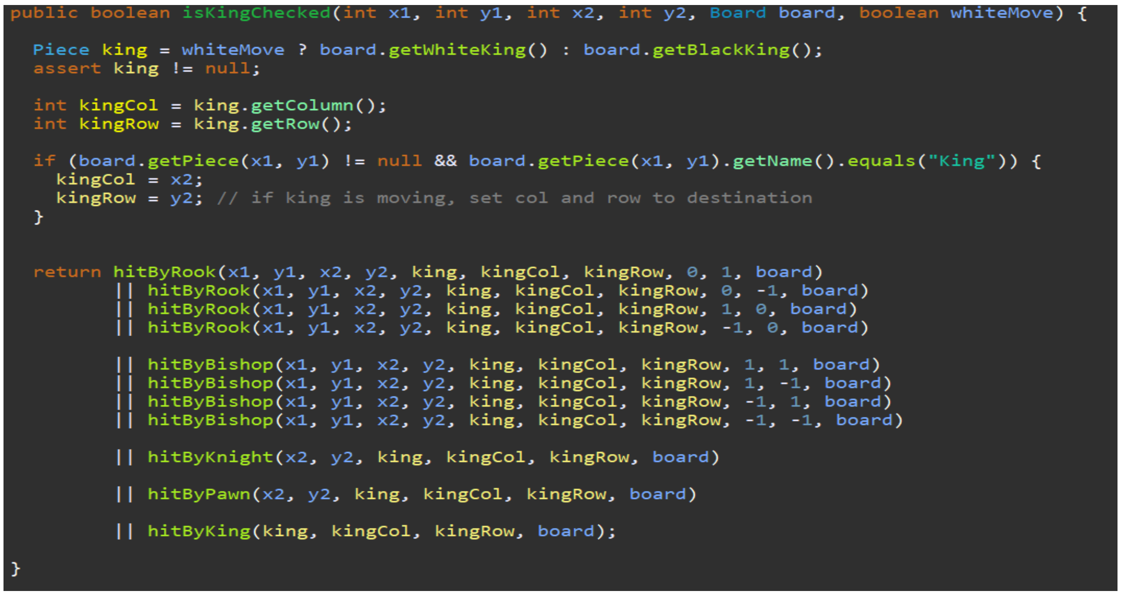
```
 board.getPiece(x1, y1).isBlack() == board.getPiece(x2, y2).isBlack();
```

If the pieces at each coordinate belong to the same team, the method will return true whereas if they differ in team, the method will return false(these pieces are already checked for null values before executing the above algorithm). Collision detection is done with the following algorithm:

```
int dx = Integer.compare(x2, x1);
    int dy = Integer.compare(y2, y1);

    int xcurr = x1 + dx;
    int ycurr = y1 +  dy;

    while (xcurr != x2 || ycurr != y2) {

      if (board.getPiece(xcurr, ycurr) != null) {
        return true;
      }

      xcurr += dx;
      ycurr += dy;
    }
    return false;
```

The dx and dy values represent the direction of movement in the x/y axis. The xcurr and ycurr values represent the current x and y value in accordance with the direction of movement. The algorithm loops through each coordinate up until the destination and checks if a piece is present in any of them. If a piece is present, the method returns true(implying that there is a collision) but if not, the xcurr and ycurr values are updated based on the direction of movement. If the loop is escaped with no colliding pieces, the method returns false.

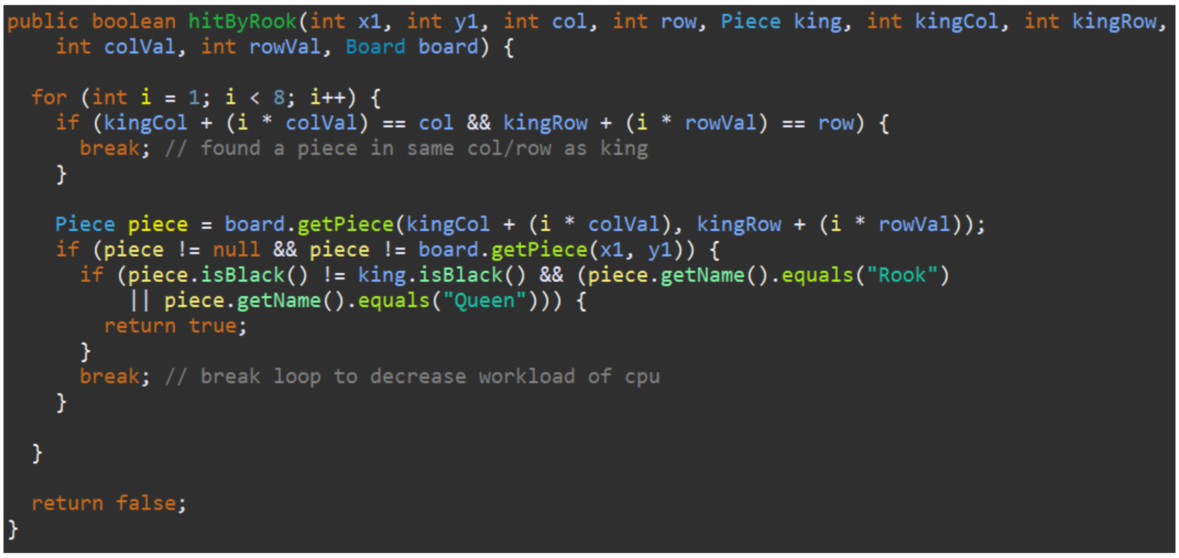
The is king checked method gets the king piece of the moving player and stores the value of the column and row. It then returns a boolean value which contains the true value if the king is in danger of being hit by any type of opponent piece as a result of the current move. It returns false if the king is not in danger after the movement happens. The reason this method cannot simply loop through each tile and see if there is a valid move landing on the king using the valid move method is because it will create an infinite loop due to the valid move method already calling the is king checked method. The following screenshot shows the entire is king checked method:



```
public boolean isKingChecked(int x1, int y1, int x2, int y2, Board board, boolean whiteMove) {

  Piece king = whiteMove ? board.getWhiteKing() : board.getBlackKing();
  assert king != null;

  int kingCol = king.getColumn();
  int kingRow = king.getRow();

  if (board.getPiece(x1, y1) != null && board.getPiece(x1, y1).getName().equals("King")) {
    kingCol = x2;
    kingRow = y2; // if king is moving, set col and row to destination
  }


  return hitByRook(x1, y1, x2, y2, king, kingCol, kingRow, 0, 1, board)
          || hitByRook(x1, y1, x2, y2, king, kingCol, kingRow, 0, -1, board)
          || hitByRook(x1, y1, x2, y2, king, kingCol, kingRow, 1, 0, board)
          || hitByRook(x1, y1, x2, y2, king, kingCol, kingRow, -1, 0, board)

          || hitByBishop(x1, y1, x2, y2, king, kingCol, kingRow, 1, 1, board)
          || hitByBishop(x1, y1, x2, y2, king, kingCol, kingRow, 1, -1, board)
          || hitByBishop(x1, y1, x2, y2, king, kingCol, kingRow, -1, 1, board)
          || hitByBishop(x1, y1, x2, y2, king, kingCol, kingRow, -1, -1, board)

          || hitByKnight(x2, y2, king, kingCol, kingRow, board)

          || hitByPawn(x2, y2, king, kingCol, kingRow, board)

          || hitByKing(king, kingCol, kingRow, board);

}
```

Figure 33. Is king checked method

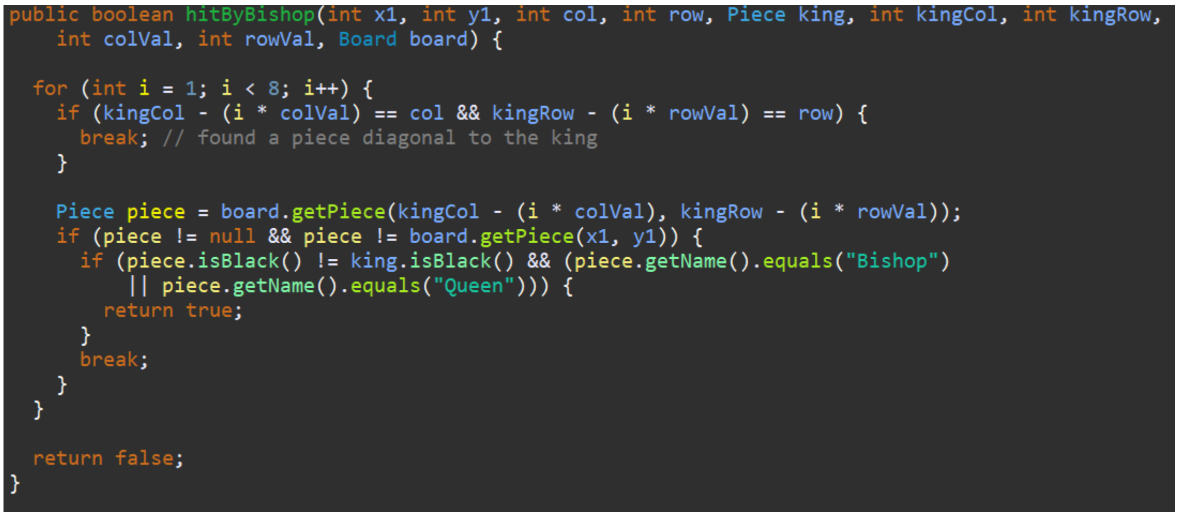
The hit by rook method performs a loop from 1 to 8 and attempts to find a piece in the same row or column as the king. If such piece exists, the method gets the piece from the board and verifies that the piece is either a rook or queen and the piece is also on the opposing team to the king. If these cases are matched, the method returns true. Otherwise it returns false. The following screenshot showcases the hit by rook method:



```
public boolean hitByRook(int x1, int y1, int col, int row, Piece king, int kingCol, int kingRow,
    int colVal, int rowVal, Board board) {

  for (int i = 1; i < 8; i++) {
    if (kingCol + (i * colVal) == col && kingRow + (i * rowVal) == row) {
      break; // found a piece in same col/row as king
    }

    Piece piece = board.getPiece(kingCol + (i * colVal), kingRow + (i * rowVal));
    if (piece != null && piece != board.getPiece(x1, y1)) {
      if (piece.isBlack() != king.isBlack() && (piece.getName().equals("Rook")
          || piece.getName().equals("Queen"))) {
        return true;
      }
      break; // break loop to decrease workload of cpu
    }

  }

  return false;
}
```

Figure 34. Hit by rook method

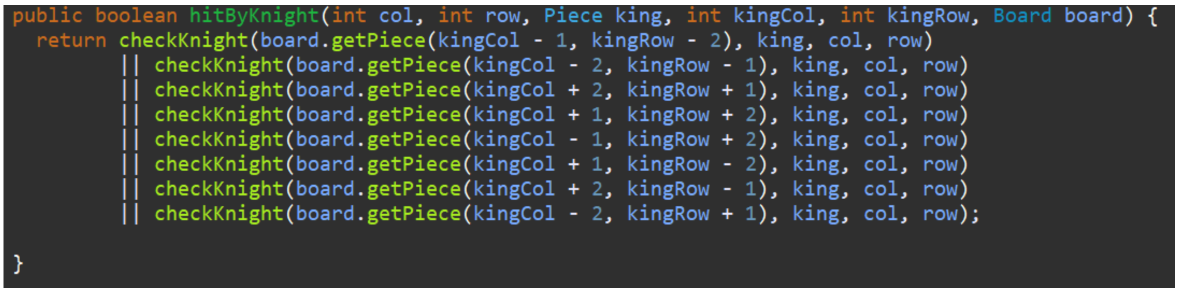
The hit by bishop method is very similar to the hit by rook method because it also performs a loop from 1 to 8 but this time it checks if there is a piece diagonal to the king. If such piece exists, the method gets the piece from the board and verifies that the piece is either a bishop or a queen and the piece is also on the opposing team to the king. If these cases are matched, the method returns true. Otherwise, it returns false. The following screenshot showcases the hit by bishop method:



```
public boolean hitByBishop(int x1, int y1, int col, int row, Piece king, int kingCol, int kingRow,
    int colVal, int rowVal, Board board) {

  for (int i = 1; i < 8; i++) {
    if (kingCol - (i * colVal) == col && kingRow - (i * rowVal) == row) {
      break; // found a piece diagonal to the king
    }

    Piece piece = board.getPiece(kingCol - (i * colVal), kingRow - (i * rowVal));
    if (piece != null && piece != board.getPiece(x1, y1)) {
      if (piece.isBlack() != king.isBlack() && (piece.getName().equals("Bishop")
          || piece.getName().equals("Queen"))) {
        return true;
      }
      break;
    }
  }

  return false;
}
```

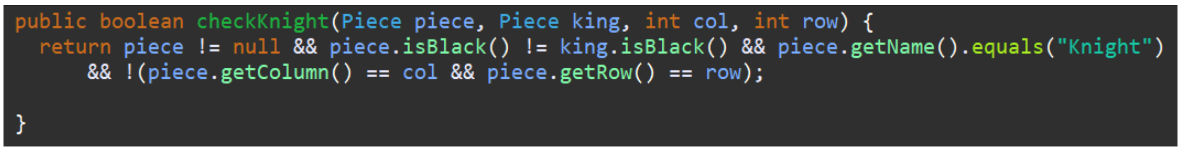
Figure 35. Hit by bishop class

The hit by knight method uses the check knight method in order to see if a knight is threatening the king. It calls the check knight method 8 times with each time changing the coordinates of the supposed knight piece. It is called 8 times because knights have a maximum of 8 potential moves per turn. The check knight method ensures a given piece is a knight and is on the opposing team to the king piece. The following screenshots show the hit by knight method and the check knight method:



```
public boolean hitByKnight(int col, int row, Piece king, int kingCol, int kingRow, Board board) {
  return checkKnight(board.getPiece(kingCol - 1, kingRow - 2), king, col, row)
         || checkKnight(board.getPiece(kingCol - 2, kingRow - 1), king, col, row)
         || checkKnight(board.getPiece(kingCol + 2, kingRow + 1), king, col, row)
         || checkKnight(board.getPiece(kingCol + 1, kingRow + 2), king, col, row)
         || checkKnight(board.getPiece(kingCol - 1, kingRow + 2), king, col, row)
         || checkKnight(board.getPiece(kingCol + 1, kingRow - 2), king, col, row)
         || checkKnight(board.getPiece(kingCol + 2, kingRow - 1), king, col, row)
         || checkKnight(board.getPiece(kingCol - 2, kingRow + 1), king, col, row);

}
```

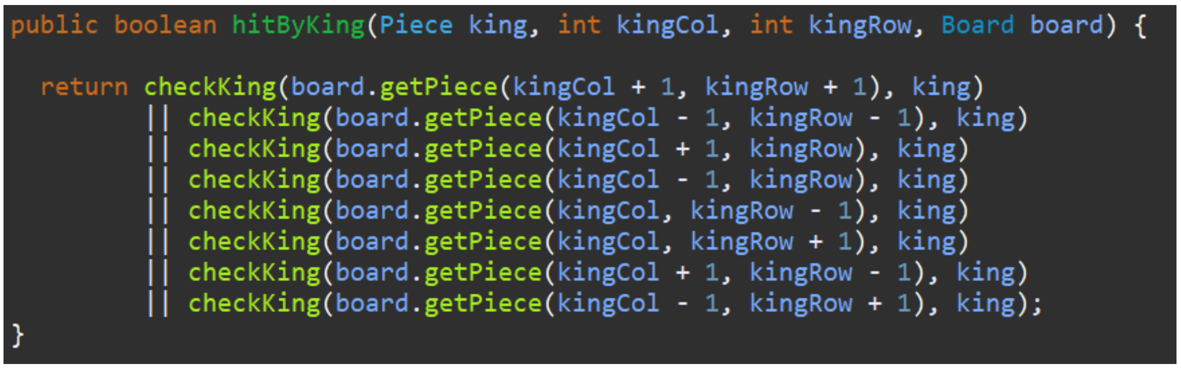
Figure 36. Hit by knight method



```
public boolean checkKnight(Piece piece, Piece king, int col, int row) {
  return piece != null && piece.isBlack() != king.isBlack() && piece.getName().equals("Knight")
      && !(piece.getColumn() == col && piece.getRow() == row);

}
```

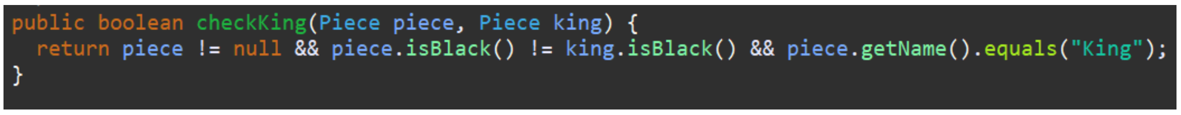
Figure 37. Check knight method

The hit by king method is very similar to the hit by knight method because it also calls its designated check king method 8 times due to a king having a maximum of 8 potential moves per round. The check king method ensures a given piece is a king and is on the opposing team to the king piece passed into the method. The following screenshots show the hit by king method and the check king method:



```
public boolean hitByKing(Piece king, int kingCol, int kingRow, Board board) {

  return checkKing(board.getPiece(kingCol + 1, kingRow + 1), king)
         || checkKing(board.getPiece(kingCol - 1, kingRow - 1), king)
         || checkKing(board.getPiece(kingCol + 1, kingRow), king)
         || checkKing(board.getPiece(kingCol - 1, kingRow), king)
         || checkKing(board.getPiece(kingCol, kingRow - 1), king)
         || checkKing(board.getPiece(kingCol, kingRow + 1), king)
         || checkKing(board.getPiece(kingCol + 1, kingRow - 1), king)
         || checkKing(board.getPiece(kingCol - 1, kingRow + 1), king);
}
```

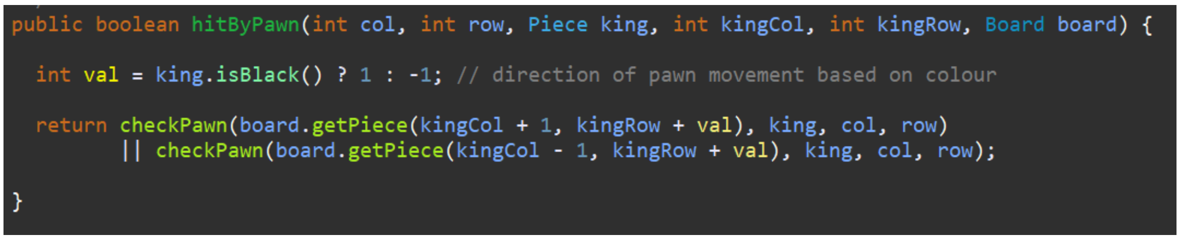
Figure 38. Hit by king method



```
public boolean checkKing(Piece piece, Piece king) {
  return piece != null && piece.isBlack() != king.isBlack() && piece.getName().equals("King");
}
```

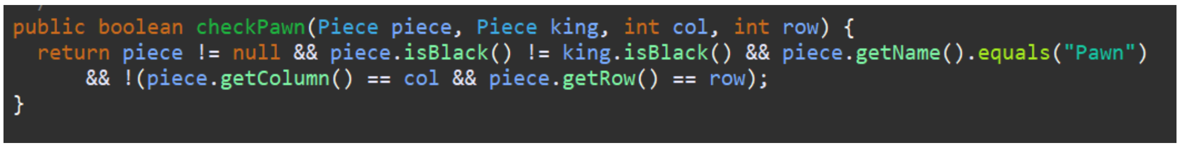
Figure 39. Check king method

The hit by pawn method first creates an integer value representing the directional value in which the pawn will move. If you are checking if the black king is in check, the integer value will become 1 because in order for a white pawn to threaten the black king, it must be in the row below it(therefore using my board setup, that is the value of the king row + 1). If you are checking if the white king is in check, the integer value will become -1.This method calls the check pawn method 2 times due to a pawn only having this many potentially threatening moves(forward left capture or forward right capture). The check pawn method ensures that a given piece is a pawn and it is on the opposing team to the king. The following screenshots show the hit by pawn method and the check king method:



```java
public boolean hitByPawn(int col, int row, Piece king, int kingCol, int kingRow, Board board) {

  int val = king.isBlack() ? 1 : -1; // direction of pawn movement based on colour

  return checkPawn(board.getPiece(kingCol + 1, kingRow + val), king, col, row)
         || checkPawn(board.getPiece(kingCol - 1, kingRow + val), king, col, row);

}
```

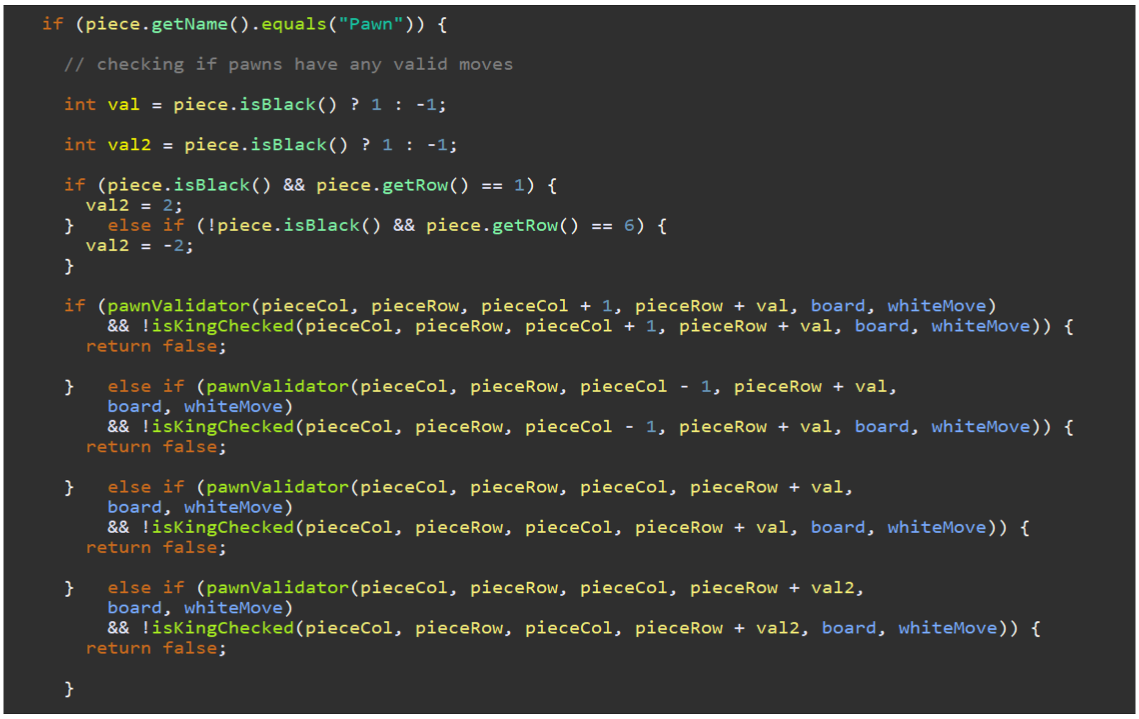
Figure 40. Hit by pawn method



```java
public boolean checkPawn(Piece piece, Piece king, int col, int row) {
  return piece != null && piece.isBlack() != king.isBlack() && piece.getName().equals("Pawn")
      && !(piece.getColumn() == col && piece.getRow() == row);
}
```

Figure 41. Check pawn method

The is king checkmated method first checks if a king is currently in check and returns false if the king is not in check due to checkmate being impossible without the king already in check. The method then stores an ArrayList of pieces based on the colour of the king. If the king is white then this method will store the white pieces but if it is black, it will store the black pieces. It then iterates through each piece and attempts to find a valid move which can take the king out of check. If any move is found, the method returns false. The method makes use of the specific piece validator for each piece while also calling the is king checked method. Although simply calling the valid move method would save a lot of lines of code due to it already validating check logic, it will result in an infinite loop in the system and therefore an approach very similar to the is king checked method was used. The following screenshot shows this method checking for a potential valid pawn movement:



```java
  if (piece.getName().equals("Pawn")) {

    // checking if pawns have any valid moves

    int val = piece.isBlack() ? 1 : -1;

    int val2 = piece.isBlack() ? 1 : -1;

    if (piece.isBlack() && piece.getRow() == 1) {
      val2 = 2;
    }   else if (!piece.isBlack() && piece.getRow() == 6) {
      val2 = -2;
    }

    if (pawnValidator(pieceCol, pieceRow, pieceCol + 1, pieceRow + val, board, whiteMove)
        && !isKingChecked(pieceCol, pieceRow, pieceCol + 1, pieceRow + val, board, whiteMove)) {
      return false;

    }   else if (pawnValidator(pieceCol, pieceRow, pieceCol - 1, pieceRow + val,
        board, whiteMove)
        && !isKingChecked(pieceCol, pieceRow, pieceCol - 1, pieceRow + val, board, whiteMove)) {
      return false;

    }   else if (pawnValidator(pieceCol, pieceRow, pieceCol, pieceRow + val,
        board, whiteMove)
        && !isKingChecked(pieceCol, pieceRow, pieceCol, pieceRow + val, board, whiteMove)) {
      return false;

    }   else if (pawnValidator(pieceCol, pieceRow, pieceCol, pieceRow + val2,
        board, whiteMove)
        && !isKingChecked(pieceCol, pieceRow, pieceCol, pieceRow + val2, board, whiteMove)) {
      return false;

    }
```

Figure 42. Pawn movement validation

Figure 42 shows 4 potential pawn movements being validated. This is due to a pawn potentially moving forwards or capturing forwards diagonally.

3.10.6 GameThread class

The game thread class is used to store information regarding game logic. The class contains two player objects which represent the players taking part in this particular game. These attributes are passed into the constructor of the class which allows different game threads to contain different players. The class also contains a board object which the game is played on. Furthermore, the class contains the singular instance of the move validator class which each game thread will use to check move validity. This move validator instance is also passed into the constructor of the class because the singular instance of the this move validator is instantiated in the driver class. The last attributes of this class contain boolean values representing which player’s turn it is and if the game is over. Due to this class implementing the Runnable interface, as soon as it is constructed and instantiated, the run method is executed. This method is used to first create the JFrame containing the game status window for this particular game. The method also displays the board and draws the pieces for each team. While the game is not over, this method is in charge of executing the game loop.

For each turn, the simulateTurn method is called which is in charge of dealing with the logic of each turn. While the board has not been pressed and released, the thread will yield itself to consume less CPU power. Once the board has been pressed and released, the method will use the appropriate functionality to form a move. This move is then passed into the validMove method in the MoveValidator instance. If the move validator returns a true value, it means the move is legal and therefore the method will update the board according to the move logic. This includes graphically moving the piece from the source coordinate to the destination coordinate and removing any pieces that may have been captured. If the move is invalid, the method checks if there is a checkmate or stalemate. If this is the case, the game will end. However, if there is not a checkmate or stalemate, the method displays a dialog box telling the user the move is invalid while also calling itself again(recursively). It does this because of the requirement of a legal move each round(unless a checkmate or stalemate ends the game).

3.10.7 Driver class

The Driver class is responsible for running the software. It first creates the singular instance of the move validator and then provides the user with an input dialog box asking them how many games they want to play(up to 4). It then uses this inputted value to iterate from 0 to the number of games and in each iteration it asks for the name of the white player and the black player. It uses these values to create a new game thread. This method is also responsible for dimensions of the interface depending on the number of games being played.

Chapter 4: **Critical Analysis**

In this chapter I will evaluate my solution to this project while also evaluating the overall project process and talking about future improvements I wish I can make on the project.

4.1 How successful was my solution

I believe my solution is very successful when it comes to functionality. This is because the game successfully implements a functioning game of chess which adheres to the rules of the game. My solution also implements concurrency in the form of multithreading which was one of the main goals I set myself for the project. Therefore, making that part of the project really successful as well. The solution also provides a presentable graphical user interface to the users and this interface also effectively provides error reports based on erroneous input. This also shows another successfully implemented goal in my project.

4.2 The project process

The project process was definitely a lot more challenging than I thought it would be before starting it. The main challenges I had were to do with time management because I needed to manage my time really effectively in order to produce a good project with the required functionality. I believe I should've planned out my management of time a lot more effectively because my original projected timeline was quite unrealistic. However, I still do believe I managed my time to a decent degree even though there have been personal issues popping up in the process hindering my ability of working on the project.

One thing I believe I did really well in this project was understand the background theory relatively quickly and to a good extent. This is due to the hours of research I did at the very beginning of the project and also during project production. This mainly involves concurrency theory but also extends to chess theory for things such as movement logic. My research of chess rules and mechanics allowed me to create a very functional game which is very user friendly and easy to play. My research of concurrency and multithreading allowed me to effectively introduce concurrency in my system and also prevented any concurrency errors during the whole development stage of the project. This is due to me picking the most effective methodology to ensure thread safety and preventing concurrency issues in my system.

4.3 Future improvements

Despite my game of chess being fully functional, there are improvements I want to make to the project in order to further expand its functionality. The first would be adding the option of playing against an AI opponent rather than just being able to play against someone else. This would greatly increase the usability of the system because it would let users play the game by themselves without the need of someone else. This would also be a very exciting concept to explore due to me personally never using artificial intelligence to that extent in a piece of software.

Another improvement I want to make is the introduction of socket programming in my system which allows the software to be hosted to a local server and accessed through different devices simultaneously. This would hugely expand the usability of the software due to it incorporating online multiplayer access rather than local play on a single machine.

Another improvement I want to make follows on from the socket programming implementation and it is the introduction of a chat room and a leaderboard. This chat room can be used to talk to other players currently on the server and it can be used to challenge other players to a game of chess. The leaderboard would be used to show which players have won the greatest number of games within the server. This would create a competitive gaming environment and encourage players to keep playing in order to climb the leaderboard.

Chapter 5: **Professional issues**

There were various professional issues that I came across in the development of my system. These vary from legal issues to ethical or management issues. These will all be discussed in this chapter.

The first issues that were encountered were plagiarism issues. These were particularly important when doing my initial research of the problem because I had to make sure I remembered where I learnt various things. This is due to the need of properly citing literature which helped with my understanding of concepts I had not known prior to starting project development. An effective citing approach was crucial in making sure the appropriate sources got their credit. I also had to make sure the code I was writing was all original code written by myself and if I did use someone else's code I would have to properly cite it in order for readers to know that I was not the author of that particular code. Plagiarism in general is unethical and also unlawful at times when things such as copyright laws are put in place. If not properly addressed, developers could face legal action and the credible sources that had their work stolen would not be credited for the amazing work they had put the effort in to making.

The second issue that was encountered was a management issue in the form of time management. Time management is a hugely important concept when it comes to developing software because time is usually limited and therefore it is required to be split up in a very effective way to maximise software development. A lacklustre use of time would likely result in incomplete code that lacks functionality and also likely includes various code smells. In my project I planned out my use of time in the early stages. This was a really good approach because it allowed me to delegate time to specific functionality at different times and I was able to effectively produce software for my project. However, I believe the planning of time I had was also a bit unrealistic due to it expecting a consistent amount of work being done each week which was not the case and should have been known to be unlikely due to myself struggling at times to manage my workload. Time management does not have any potential legal issues but bad time management can really slow down the production of software and bad time management in collaborate development can also be seen as unethical.

The next issue that was encountered in my project was the issue of usability in my system. I decided early on in my development that I wanted to create a very usable piece of software and therefore I needed to provide certain functionality that did this. This also explains the goal of providing a nice graphical interface with error reporting in my introduction. My system contains a lot of graphical error reporting for things such as invalid moves and I believe my graphical interface as a whole is very easy to use. There are features I could add to my software in order to increase the usability. These include things such as highlighting valid moves in a different colour when a piece is clicked or providing users with a rule book for the game of chess. These features would be particularly useful for newer players of chess who do not remember the rules off by heart. Usability is a really important factor to consider in development because you want software to be as user friendly as possible in order for more people to use it and enjoy it. If a piece of software is too complex and does not provide users with instructions, they may be discouraged from using the software again.

In general, when professional issues are not met, there can be a range of issues. For example, as mentioned before, the use of plagiarism can result in legal action being taken place. Developers should try their best to avoid professional issues in development. This will almost certainly allow them to develop very powerful software that is very usable and very functional.

Bibliography

[1] Baeldung. “Process vs. Thread” 2023. [Online]. Available: https://www.baeldung.com/cs/process-vsthread#:~:text=A%20thread%20is%20a%20semi%2Dprocess.&text=Unlike%20a%20real%20process%2C%20the, level%20threads%2C%20and%20hybrid%20threads.

[2] Vivek, Satyam. “Threads in Operating System” 2022. [Online]. Available: https://www.scaler.com/topics/operating-system/threads-in-operating-system/

[3] Chilkuri, Dinesh. “Advantages and Disadvantages of Threads”. [Online]. Available: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/Advantages%20and%20disadvanta ges.html

[4] Ungerer, Theo, Borut Robič, and Jurij Šilc. "Multithreaded processors." The Computer Journal 45.3 (2002): 320-348.

[5] Walilko, Andrej. “What are CPU Cores vs Threads” 2023. [Online]. Available: https://www.liquidweb.com/blog/difference-cpu-coresthread/#:~:text=Modern%20processors%20support%20hyperthreading%2C%20a,available%20resources%20and %20increasing%20throughput

[6] Pankaj. “Multithreading in Java – Everything you MUST Know” 2022. [Online]. Available: https://www.digitalocean.com/community/tutorials/multithreading-in-java

[7] Bloch, Joshua. Effective java. Addison-Wesley Professional, 2008.

[8] Zhang, Charles. "FlexSync: An aspect-oriented approach to Java synchronization." 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009.

[9] Parallel Programming. “How do you compare and contrast different lock algorithms and data structures in parallel programming?”. [Online]. Available: https://www.linkedin.com/advice/1/how-do-you-compare-contrastdifferentlock#:~:text=This%20is%20where%20lock%20algorithms,from%20interfering%20or%20causing%20conflicts

[10] HaiyingYu. “Race conditions and deadlocks” 2022. [Online]. Available: https://learn.microsoft.com/enus/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks

[11] MIT. “Reading 23: Locks and Synchronization” 2015. [Online]. Available: https://web.mit.edu/6.005/www/fa15/classes/23-locks/#goals_of_concurrent_program_design

[12] David Rice and Matt Foemmel. “Coarse-Grained Lock”. [Online]. Available: https://martinfowler.com/eaaCatalog/coarseGrainedLock.html

[13] Pankaj. “Java Singleton Design Pattern Best Practices with Examples” 2022. [Online]. Available: https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples

Appendix: Project diary

17/10/23: Finished adding pieces for both teams.

26/10/23: Stopped coding because I realised my planned class design has too many dependencies and is not done well. I have now thought of a new class design plan which I am going to implement.

26/10/23: Finished refactoring old code so it follows new class design plan.

14/11/23: Players can move in a turn based way. Need to start work on MoveValidator before refactoring gameThread.

17/11/23: Added all basic movement logic to move validator class. In doing so, I noticed that the work I did on 14/11/23 may not have been the most relevant at the time because a lot of it will be replaced.

19/11/23: Added collision detetction for each move.

20/11/23: Added Capture logic to GameThread. Need to go back to work on moveValidator to implement more complex captures like pawn captures.

21/11/23: Added Pawn capture logic. Next thing to add will be check/checkmate logic. This will require refactoring the moveValidator class.

11/03/24: Added Pawn promotion logic. Currently stuck on implementing a choice factor into it so th system currently only promotes to a Queen.

19/03/24: Added Check and checkmate logic. Struggling to find a way to allow gameThread to recognise a checkmate.

04/04/24: Added new GUI to each game for showing game status. Also implemented pawn promotion properly.

08/04/24: Finally completed adding checkmate logic. It was very difficult to implement at first because every method i tried did not work but I found a way to do it.

08/04/24: Added stalemate logic and now the code is fully functional. The game can end in either a win or a draw.