A concurrency based game environment

Full Unit Project

Interim Report

Mohammed Zuhayr Mohsen

December 2023

Supervised by: Zhaohui Luo

Department of Computer Science

Royal Holloway University of London

Contents

Introduction 3

Advantages of Threads 3

Multithreading 3

Risks of Multithreading 3

Preventing concurrency issues 3

Synchronization using locks 3

Synchronization using a singleton 4

The Bill Pugh singleton 5

Concurrency in a game environment 5

Literature review 7

Aims and Objectives 8

Multithreading 8

Graphical User Interface 8

Socket/Network programming 8

Summary of completed work 9

Movement mechanics currently in my game 9

Capturing mechanics currently in my game 9

Missing game functionality currently in my game 9

Pawn promotion logic 9

Check/Checkmate logic 9

Win/Loss/Draw logic 9

User interface currently in my system 10

Class Design 10

Gui classes 11

Piece classes 11

Player classes 11

Move class 11

MoveValidator class 12

GameThread class 14

Class dependencies 15

Testing strategy 15

Planning and time scale 16

Reflecting on my first term plan 16

Term 2 plan 16

Bibliography 18

References........18
Project Diary........19

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]. 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 Threads executing the same instruction to not always return the same value (if local variables are manipulated in different ways). 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.

Advantages of Threads

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.

Multithreading

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.

Risks of Multithreading

Despite the numerous advantages of Multithreading, there are also some potential risks that need 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.

Preventing concurrency issues

Synchronization using locks

There are a number of ways to prevent potential 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]. 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.

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.

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’ inner 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.

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.

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 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 another 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 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.

These were the main references that allowed me to fully grasp the concept of concurrency and also how to implement it in a practical way.

Aims and Objectives

The main aim 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.

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.

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. 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. To prevent any possible movement errors, the graphical interface for each game can highlight each tile a piece can move to when selected.

Socket/Network programming

In order to achieve a multiplayer environment, the project will ideally implement client-server communication with the help of socket programming [14]. Socket programming is used to connect Nodes on a network together and it can be used to connect users to the hosted game environment. Socket programming can be used to locally host the game environment by combining input and output streams [15] in the server and clients. Locally hosting the environment will allow clients on multiple different machines to access the system and it can allow them to challenge each other or view what is happening in other games. In extension, a client server architecture can allow connected clients(users) to communicate via a chat room. The client-server hosting will be very crucial for allowing multiple users to access multiple games and it will also make the system a lot more usable due to it not being restricted to a singular machine.

Summary of completed work

Currently, the system supports a turn based game of chess between two players. One player has each of the white pieces whereas the other has each of the black pieces. Each player is able to move at most one of their pieces per turn and each piece is able to move according to its specified movement mechanics.

Movement mechanics currently in my game

Pawns are only able to move forward, the first time a pawn moves it is able to move up to two tiles, otherwise it may move up to one tile forwards. 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 unless that piece is captured and the tile is vacated. 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 can not 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 can not 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.

Capturing mechanics currently in my game

Capturing has been implemented in my current system. 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. 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.

Missing game functionality currently in my game

Pawn promotion logic

Currently, when a pawn makes its way over to the complete other side of the board, it is unable to move any more due to it only being able to move forward. Since there are no more tiles currently ‘ahead’ of the pawn, it is forced to stay where it is for the remainder of the game.

Check/Checkmate logic

Currently, my system does not support any form of check/checkmate logic. Kings in my current system are able to be captured as if they were any other piece. This also means that my system does not restrict a kings movements to avoid a potential check/checkmate.

Win/Loss/Draw logic

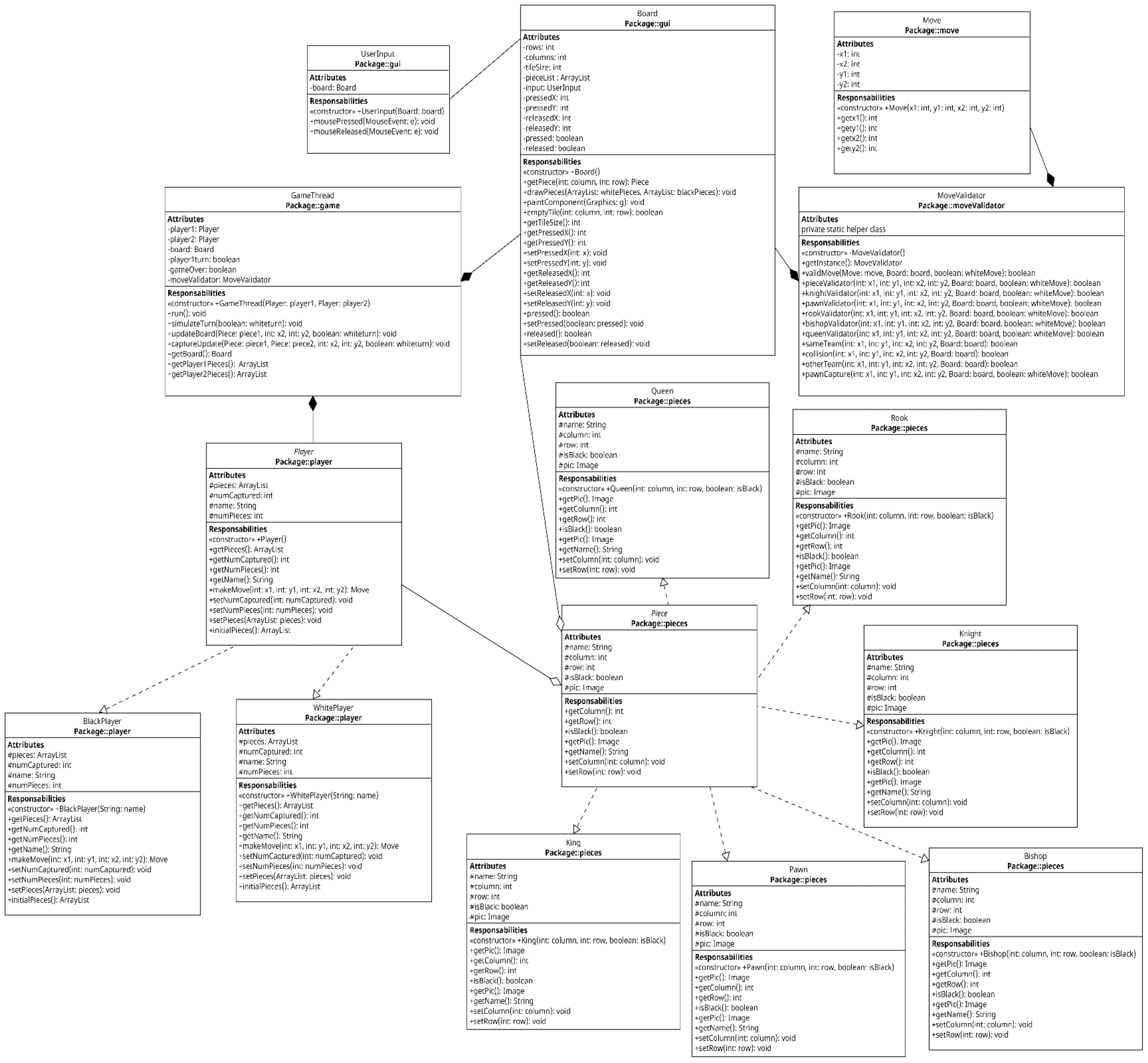
Currently, my system does not generate any form of outcome when it comes to the chess game. This is mainly due to the fact that check/checkmate logic has not been implemented, therefore no player will be able to be crowned a winner. My system allows each player to capture as many pieces as possible of the opposing team but even if this is done, the game will not end.

User interface currently in my system

The system currently contains a graphical user interface made with the swing toolkit. Currently, the interface is only big enough to fit one game of chess due to my current system not supporting multiple games. The chess board contains a primarily blue combination of tile colours and pieces of each team are represented as image sprites. The interface is very clean and easy on the eye.

Class Design

The following UML represents the class design and dependencies in my system.


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
+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
UserInput
Package::gui
Attributes
-board: Board
Responsabilities
«constructor» +UserInput(Board: board)
+mousePressed(MouseEvent: e): void
+mouseReleased(MouseEvent: e): 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)
-run(): void
-simulateTurn(boolean: whiteturn): void
+updateBoard(Piece: piece1, int: x2, int: y2, 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
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
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
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
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
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
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
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

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.

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.

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 and 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.

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 as long as 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.

MoveValidator class

The MoveValidator 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. Below is a code snippet showcasing the Bill Pugh singleton design of this class.

```
public class MoveValidator {

  private MoveValidator() {}

  private static class Helper {
    private static final MoveValidator INSTANCE = new MoveValidator();
    // singleton class
  }

  public static MoveValidator getInstance() {
    return Helper.INSTANCE;
    // class only instantiated when this is run
  }
```

ValidMove method

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. 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 pieceValidator method. This 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.

KnightValidator method

The knightValidator 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.

RookValidator method

The rookValidator method returns a true value if the rook moves either vertically of 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.

BishopValidator method

The bishopValidator 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. However, this line of code would allow a movement in which the x and y values do not change, therefore the method requires an additional line suppressing this possibility.

QueenValidator method

The queenValidator method returns a true value if the movement satisfies that of a bishop or a rook. It does this with the following line of code: bishopValidator(x1, y1, x2, y2, board, whiteMove) || rookValidator(x1, y1, x2, y2, board, whiteMove).

KingValidator method

The kingValidator 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.

PawnValidator method

The pawnValidator 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 algorithm is used to determine 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
```

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 pawnCapture method.

PawnCapture method

The pawnCapture 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.

Collision and SameTeam methods

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 to 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.

GameThread class

The GameThread 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 MoveValidator class which each game thread will use to check move validity. 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 draw the pieces onto the board and contains 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 methods to form a move. This move is then passed into the validMove method in the MoveValidator instance. The following shows how this method acquires the movement data:

```
int x1 = this.getBoard().getPressedX();
int y1 = this.getBoard().getPressedY();
Piece piece1 = this.getBoard().getPiece(x1, y1);
if (this.getBoard().released()) {
  int x2 = this.getBoard().getReleasedX();
  int y2 = this.getBoard().getReleasedY();
  Piece piece2 = this.getBoard().getPiece(x2, y2);
  Move move = new Move(x1, y1, x2, y2);
```

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 simulateTurn method will call itself again. It does this because of the requirement of a legal move each round.

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 can not 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 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.

Testing strategy

My code has followed a test driven development approach. 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.

Planning and time scale

When planning the project a couple months ago, I wanted to focus heavily on the functionality of a singular chess game in first term. Only once a game was fully functional, I would start adding multiple games and include any socket/network programming(in term 2).

Reflecting on my first term plan

According to my original plan, at this point in time I should have a fully functional chess game which can then be transformed into multiple games at the start of second term. However, as shown in the previous section, this has not been the case. When I started coding the system, my proposed class design contained a large number of dependencies that I did not suppress. Therefore, since I had already implemented this class design in the first few weeks of programming, I had to then stop worrying about functionality and start worrying about reducing the dependencies in the system. This refactoring of the class design took away valuable time, which could have been spent on functionality, however it was crucial in preventing any possible future data corruption when I started implementing Threads. Despite taking away time, this refactoring was done at a very good time because the system lacked a lot of functionality and was therefore easier to refactor. Despite the huge refactoring carried out on the system, it still contains the majority of the functionality needed to make it a fully playable game of chess. Therefore, I am not very far behind my proposed plan for first term.

Term 2 plan

My plan for second term is quite similar to what I proposed in my original plan, however the functionality of the system will still need to be worked on at the start of term. Furthermore, due to my knowledge of potential concurrency/network issues, I have tried to make my second term plan more sensitive to any potential issues and give myself more breathing room to fix any said issue.

The following is my plan for term 2:

Week starting 08/01/24: Implement pawn promotion logic and start work on check/checkmate logic

Week starting 15/01/24: Continue implementing check/checkmate logic

Week starting 22/01/24 + 29/01/24: Implement multiple games and work on inter-thread communication which allows multiple game threads to make use of the move legality validation module

Week starting 05/02/24: Create sockets for server and clients and establish connection

Week starting 12/02/24 + 19/02/24 + 26/02/24: Allow locally hosted game server to host multiple clients and let each play against each other

Week starting 04/03/24: Implement chat room and make touch ups on graphical interface. This includes implementing any graphical error reports

Week starting 11/03/24 + 18/03/24: Prepare for final report

I believe the above plan is very realistic and still provides myself with potential breathing room in case of any unknown issues that may occur. I believe there may be issues when it comes to implementing socket programming due to the fact that I have never attempted it at such a large scale in Java. This is the reason I allowed myself three weeks to get the game server up and running and to make sure it allows each game to be played exactly as it should. I would hopefully like the code to be completed by the time the week starting 11/03/24 commences because it allows me to carry out vigorous system testing to exploit any unseen bugs. There should be very minimal/none of these bugs due to the test driven development being implemented in the system but this time frame allows plenty of time for any issues to be resolved.

Bibliography

References

[1]: Baeldung. “Process vs. Thread” 2023. [Online]. Available: https://www.baeldung.com/cs/process-vs-thread#:~: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%20disadvantages.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-cores-thread/#:~: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-contrast-different-lock#:~:text=This%20is%20where%20lock%20algorithms,from%20interfering%20or%20causing%20conflicts

[10]: HaiyingYu. “Race conditions and deadlocks” 2022. [Online]. Available: https://learn.microsoft.com/en-us/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

[14]: Kalita, Limi. "Socket programming." *International Journal of Computer Science and Information Technologies* 5.3 (2014): 4802-4807.

[15]: Mahmoud, Qusay H. "Sockets programming in Java: A tutorial." *JavaWorld Online Journal* (1996).

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.