Final Report

Draughts Game

Group 4

Abhishek Nandi
Christian O'Connell
Roston Pereira
Callum Stocker
Le Wang-Riches
Piers Williams

CE903 Group Project

Dr Paul Scott

19th March 2014

Contents

List of Figures

1.0 Introduction

The goal of this document is to provide the stakeholders a complete documentation report on the project completed by Group 4. The project is to complete a Draughts game for CE903. The report is broken down into several key sections for ease of access and reading; the first section is the introduction and will outline the project's goal and methodology.

The next section is System Design which will provide readers with an in depth review of the actions this group took in order to complete the project, in particular the design decisions. This section will refer to the Software Requirement Specification [1].

The implementation section will cover the code implementation the explanation of the code, the language used and the key components. Potential problems we encountered and the overview of the classes and main functions. Testing is after the implementation and will have the test this group used and decisions for these tests, how they were designed and the results of each test conducted by the group.

Finally, the conclusion will discuss the group's successes or failures, and possible improvements with future iterations. Other items will be addressed; the tools and methodology the effectiveness or how appropriate were these in this project, and how the project was managed.

The biggest challenge this group encountered during the project implementation was working in a group. Team work introduces more challenges, such as integration, communication and development, especially when designing key component to work together.

1.1 Abbreviations, Acronyms and Definitions

Artificial Intelligence (AI) – The development and research into the field of a computer system to mimic human intelligence, in order to perform a variety of tasks, similar to a human.

Java – A high level object oriented programming language, the main language used to implement the game of Draughts.

Graphical User Interface (GUI) – Is a type of user interface which allows the user to interact with the system using Icons, Menus and Buttons, as oppose to use of text-based interface or type command (command line). This allow all level of computer user to able learn and quickly use a system without train.

User Interface – is a system which the human use to interact with a machine (similar to GUI)

Extreme Programming (XP) – Is a software development methodology intended to improve software development using a number of practises, such as test-driven development or pair programming. These practises aim to overcome the problem traditional software development faces.

Java Virtual Machine (JVM) – is used interpret Java binary code (or byte code) for a computer's processor so it can use the Java programming instructions. Thus Java was designed to be able to run on virtually any platform.

Subversion (SVN) – Version control for software integration for users in different geological locations.

Unified Modelling Language (UML) – A modelling language used in field of software engineering, it provide graphical visualisation of the software information, its design and characteristic of the software.

Heuristic – is to find or discover, it is a process of gaining knowledge through experience to solve problems.

Intelligent Agent – Software which makes decisions on its environments through sensors and act on its actuators or sensors.

Software Requirement Specification – The document containing the requirements specification for a project

IntelliJ – is a Java cross platform Integrated Development Environment, and is the main tool used in the development of this project.

1.2 Scope

The project is to implement Draught Game (Checkers), where a human player can play against an Artificial Intelligent player using a Graphical User Interface with a variety of changeable rule sets. There are a number of additional features to improve the overall user experience. The project is designed to entertain the user with a game of Draughts.

1.3 Project Goals

The main goal of the project is to implement a Draughts game, with a few additional features. The main challenge this project encountered was during the implementation; especially working to smoothly integrate all the components to the project. Pair programming was employed with each pair addressing a different component, a common extreme programming practise of pair programming. Different groups worked on different part of the Draughts game. Often, this is greatest challenge when working in teams, often teams could drift apart with assumptions due to poor communication between teams.

1.4 Project Methodology

The decision was made early in the project to use the agile Extreme Programming development methodology [2]. The decision was made based on a number of contributing factors including its principles for group cohesion to overcome problems when working in a team as well as several members having prior experience with the extreme programming methodology. During the implementation stage of the assignment we regularly employed extreme programming practices to reduce the chance of potential problems that would arise in a team based project.

For example, one of the biggest challenges the team would face is work integration, which meant the teams had to cooperating in order to smoothly integrate each component from time to time. We mitigated this potential problem by ensuring our teams had regular team meetings so all members were aware of the each other's work. Teams would also work together to ensure integration were done on only one version in the SVN to avoid clash in versions. This was but one of many practices we employed during the implementation period of the project. Below are a short list of practises and our experiences.

One of the early practices we employed was the use of two iterations, separating the workload into core requirements, and additional, allowing shorter more focuses goals which could be addressed during regular team meetings, thus we were able to complete the first iteration 5 days before the scheduled date. In addition, pair programming was introduced to reduce bugs and allows other programmer to be aware of others work, thus at later day, code integration can be done smoothly. During the implementation we also took advantages Code Stand and System Metaphor making the software much more readable, and making the code easier to transverse. The concept of collective ownership was also practiced, whereby no team member held claim to individual sections of code, rather the entire team 'owned' what had been produced. This aimed to improve integration of sections, as well as encourage the fixing of bugs when seen, rather than waiting for the original author to fix them, slowing down the project schedule.

1.5 Project Development Tools

The main tool used in the development was the IntelliJ Integrated Development Environment (IDE), as all members had prior experience with IntelliJ, as well as its support for version control such as subversion. Thus it was a logical choice in the development to use this for the development tool. It features number of advanced code navigation and refactoring capabilities which all team members are familiar with. The team also used Microsoft Visio to create the UML diagrams.

The desired platform for the Draughts game to work on is on desktop computer which has latest JVM installed. The team had no problems working with either tools, no complaint was ever lodged from any team members during meetings. All members are happy and satisfied by the tools used in this project, and if we had to improve on the project for future iteration the team would be agree on the same set of tools and methodology.

1.6 Document Overview

The following document can be broken down into four sections, each respective to different part of the project, the System Design on the design approach and decisions. The System Design is meant for readers who are keen design consideration this group used when creating the Draughts game. The Implementation is on the actual system and the key components for readers whom are interested how the Draughts work. The Implementation will also cover code and programming languages used in this this project, a brief summary of their effectiveness will also be covered.

The test were conducted by team members on each sections, see Testing for details. Finally the conclusion details the successes and failures for the project, and considers the possible reasons for this.

The conclusion will provide an overview on all aspect of the project. From tools, design, overall effectiveness of the Team, Methodology to project management.

2.0 System Design

2.1 System Requirements

The requirements were a Draughts game against an Artificial Intelligent opponent, for requirements in detail see the Software Requirement Specification [1]. The requirements can be grouped in three categories: the game logic, the user interface and the opponent. Each category will be briefly mentioned and discussed, some categories will overlap with each other for example, and rules of the board and User Interface on the board will have the same requirements.

The rules of Draughts [3] are one of the main requirements for our system, the requirement cover the pieces and the rules. The board requirement will be covered in the sections below. The requirements were for two pieces, the standard man and the king. Both must be represented on the board in white and black colour by default, each piece will have similar and different movement requirements. The standard man will be able to move diagonally forward once but can "take" diagonally forward multiple times; it may only take the opposite colour. They may not move onto a square occupied by another piece or move backwards. The King can do everything the standard man can; it can also "take" and move backwards diagonally. The requirement should also state, "if" a standard man reach the other side of the board (the last row) that standard man will become a king.

The game will be by default make black going first but this will be changeable in the options and as will some rules of the game. Since Draughts has different rules base in different geological location, there will be some minor rule option which the player will be able to change. For example, in some Draughts the standard Man can captures backward or long-range kings allows the king move more than one space per turn. "Must capture" rules stats capture must be made if possible, even at the disadvantage of the player. International Draught has a different set of rules, and conforms to a standard and is widely recognised [4].

The User Interface designs were with a "windows" and within this is a Draughts board, with menus and icons at the top of the window. The board by default will have 8 by 8 squares with black and white colour. These settings can be changed to 10 by 10 board and the piece's colours can be changed by the players. The menu requirements are there should be options:

1. New game (starts a new game)
2. Redo Last move (cancels player's last move)
3. Hints (provides a suggest to player where to go next)
4. Options (a panel for all the option)
5. Close (close the program)

The Option should hold the colours options and settings for rules and the AI difficulty. The colours of the pieces and boards should be changeable and can be set by the player in the options panel too.

The AI will be the player's opponent when the user is playing our Draughts game. The AI's difficulty can be changed depending on the skill level of the users. The AI should offer a moderate challenge to casual

players. The performance requirement is also taken in account since the Draughts is a solved game [5]. However, for an AI to search for all the possible moves, the number of board moves would be around 10^{20}. Making Draughts a solved game with the largest search space requirements, thus it is not possible for a computer calculate the possible moves in a practical time. The requirement also needed a heuristic search to help the agent to make the most logical requirements without the need to virtualise the entire search space. The requirement is that no matter the difficulty the all moves by the AI should be in a short space of time.

2.2 System Architecture

The system can be broken down into three key components, the Artificial Intelligence, Core Board Logic and the Graphical Interface. These three makeups the entire Draughts game, where all logic and algorithm of the code reside. Figure 2.1 shows the use case diagram.

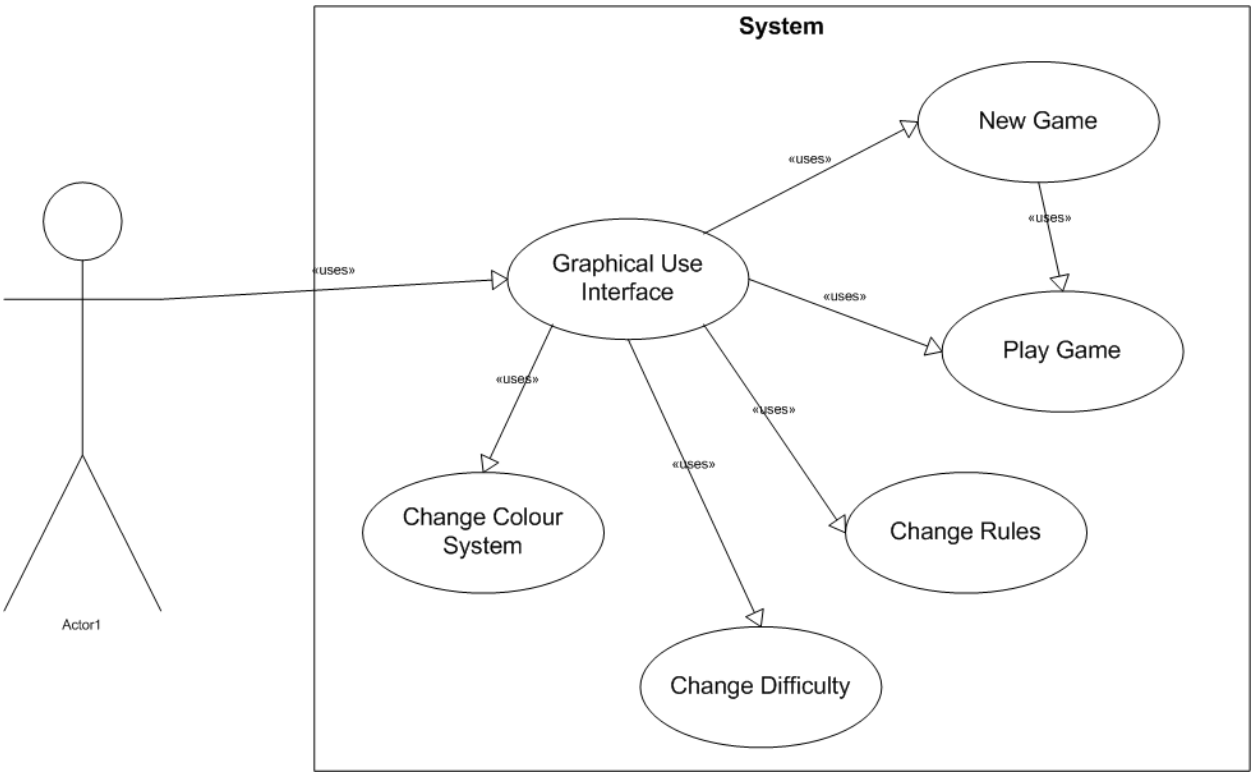


Fig. 2.1 Use Case Diagram
Adapted from Figure 2.1 in [1]

The use case above is slightly changed from the SRS use case; see the SRS for full detail. There were very little changes to the Use case diagram, the only major change was the player no longer needed access to the Core Logic all options are in the GUI. The GUI design is shown in Figure 2.2.

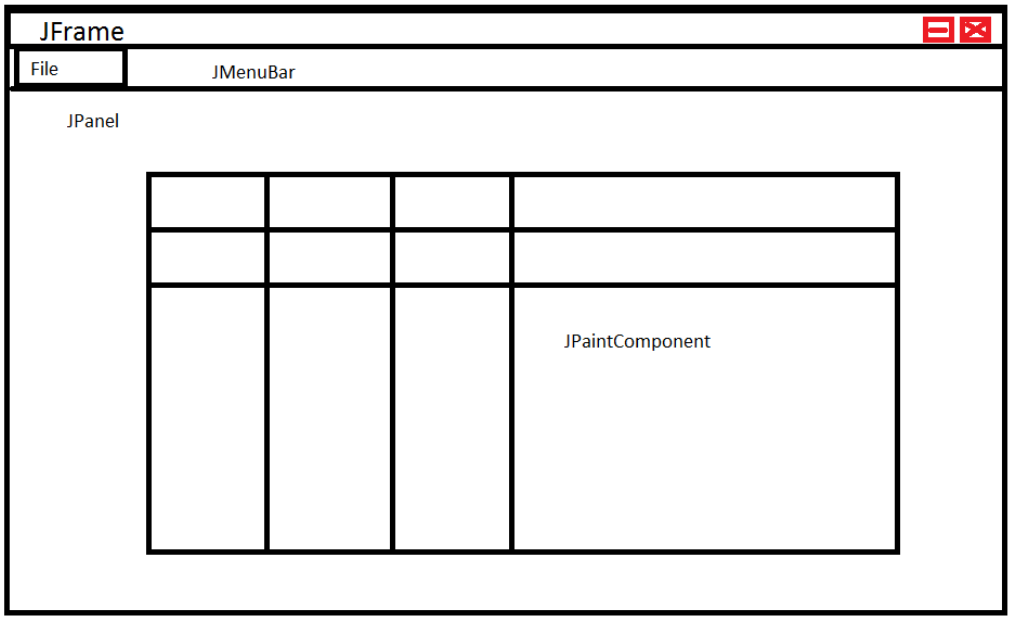


Fig. 2.2 The basic design

A Tree representation of the Graphical User Interface is shown in Figure 2.3.

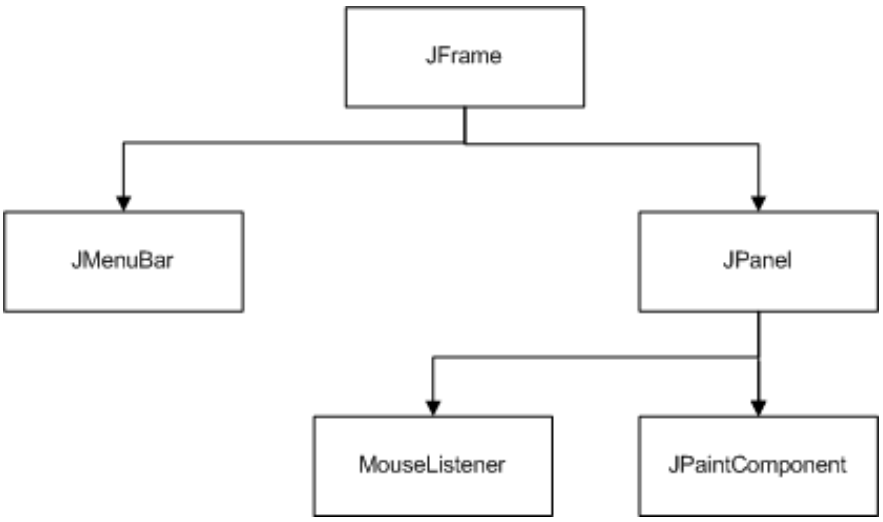


Fig. 2.3 A Tree Diagram of the GUI

The design of the GUI changed very little, and remained the same as outlined in the SRS document.

2.3 Main Components

The main components to the Draughts game consist of the three packages contain the AI, User Interface and the Core Logic. The GUIs main role is drawing the current state of the board and to highlight potential moves for the user; it will also display the various menus and icons the user can use. The core logic holds the information regarding the current state of the game and manages the transition between states according to the game rules and inputs from the user. The AI is responsible for supplying counter moves to the human player using information regarding the state of the board to make an apparent informed choice with different levels of difficulty.

The original system from SRS was designed so that the user would be able to play the game using the GUI which uses the Core Logic as a bridge between the AI and GUI. Thus the Core Logic would handle the data flows. The current system changed very little from that design. During Iteration 1, the GUI handled the main loop of the game to allow the entire process to be driven by user input events for ease of development to achieve the goals of the first iteration. This was rectified in iteration 2 however when multithreading was introduced and the main loop was handled separately to the Event Dispatcher Thread, reverting to the original design. The current state of the components are when a user select a piece, GUI calls the core logic to check if the move is valid. After that if the user makes a move, the Core logic collects all the possible moves and passes this onto the AI, for the AI's turn. The working of the game can be better understood by looking at the Data Flow Diagram in Figure 2.4. The Data Flow Diagram clearly shows the core logic as an intermediately for the player and the AI opponent.

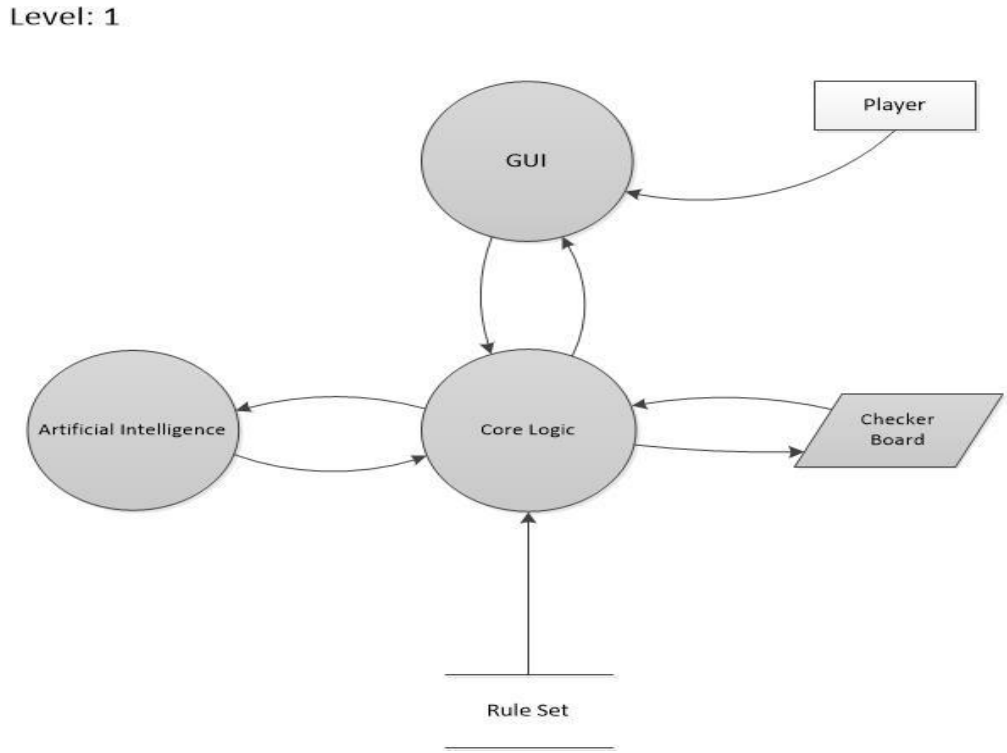


Fig 2.4 Data Flow Diagram
Adapted From Fig 2.4 in [1]

2.4. Hardware and Algorithm

This project is required to run on a standard desktop computer with a Java Virtual Machine installed. The machines in the university laboratory were used for the test standard.

3.0 Implementation

3.1 Architecture

The architecture for the implementation was based upon the *Model View Controller* (MVC) architecture, as the three elements: the *Model* (how it works), the *View* (how it looks) and the *Controller* (how it interacts), corresponded identically with the natural segmentation of the core systems, the core logic, the GUI, and the AI opponent respectively. The aim of keeping these items separate was to allow each sub-team of the group to implement their given sections in parallel before being integrated. The final architecture is shown in Figure 3.1, where the class diagram has been generated.

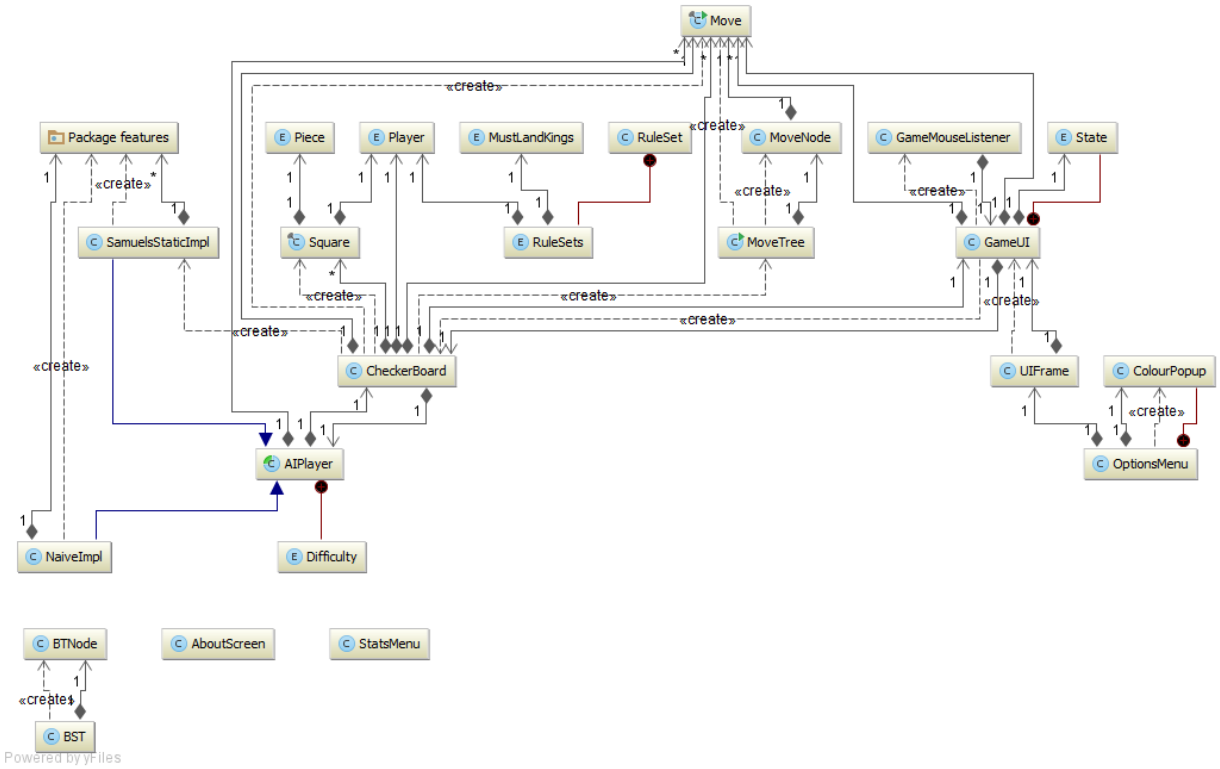


Fig. 3.1 Class Diagram Generated By IntelliJ
(Generated by JetBrains IntelliJ 13.0.1)

3.2 Core Logic

3.2.1 Classes

The Piece Enum (Table 3.1) records the value of the piece (if any) present in a square.

Value	Meaning
Empty	The field does not contain a piece
StandardMan	The field contains a piece that is not a king
King	The field contains a piece that is a king

Table 3.1 Piece Enum

The Player Enum (Table 3.2) refers to the human player as White, and the AI as Black, regardless of their actual display colour for ease of development purposes.

Value	Meaning
Black	The Player is playing with the Black pieces
White	The Player is playing with the White pieces

Table 3.2 Player Enum

The Square class (Table 3.3) represents a single cell on a checker board. *Squares* are private and cannot be constructed outside of the Square class. To provide the necessary *Squares* to the *CheckerBoard*, there are a number of public static fields that contain helpful Square instances.

Field	Type	Meaning
player	Player	The Player this square belongs to
piece	Piece	The Piece that this square contains
value	int	The value of the square to the AI

Table 3.3 Square

The RuleSet Enum (Table 3.4) is singly responsible for storing the various possible *Rule Set's* for the game, as well as their individual characteristics. The fields are:

Field	Type	Meaning
displayName	String	Name of the Rule Set for use in the UI
firstMove	Player	The player that moves first in this game
boardSize	int	The size of the board in both Width and Height
menCaptureBackwards	Boolean	Can men capture pieces backwards?
longRangeKings	Boolean	Can kings move in the long range fashion
mustLandKings	MustLandKings(Enum)	See Below
startingPieces	int	Number of pieces per player on the board

Table 3.4 Ruleset

The MustLandKings Enum (Table 3.5) stores the possible values for its corresponding value in the RuleSet Enum.

Value	Meaning
True	A standard man must land on the kings row and stay there to be promoted
False	A standard man must touch the kings row during a single move or land there to be promoted
TurnStop	If a standard man touches the kings row it will be promoted and the turn will be forced to stop

Table 3.5 MustLandKings

The CheckerBoard class (Table 3.6) is the main class that represents that game of Draughts itself. *CheckerBoard* holds a number of fields for achieving its responsibility. The main jobs that this class performs is to store the current state of the board, track whose turn it currently is, store the previous state of the board for the undo action determine the legal moves for a selected piece. These are elaborated upon in the UML diagram in Figure 3.2.

Field	Type	Meaning
board	Square[][]	A two dimensional array holding Square objects
size	int	The size of the board, needed for array index operations
settings	Settings	Stores various settings for the game
previousMove	Move	The Move that was used previously
currentGo	Player	The Player whose turn it is now
nextGo	Player	The Player whose turn it is next, after the current go
opponent	AIPlayer	A link to the AI, allowing the AI to be called when it is its game
ui	GameUI	A link to the UI for doing things
pMove	LinkedList<Move>	List of Moves ... Why?
previousState	CheckerBoard	The previous state of the board, allows Un-doing
isRun	boolean	True when the game's main loop is running

Table 3.6 CheckerBoard

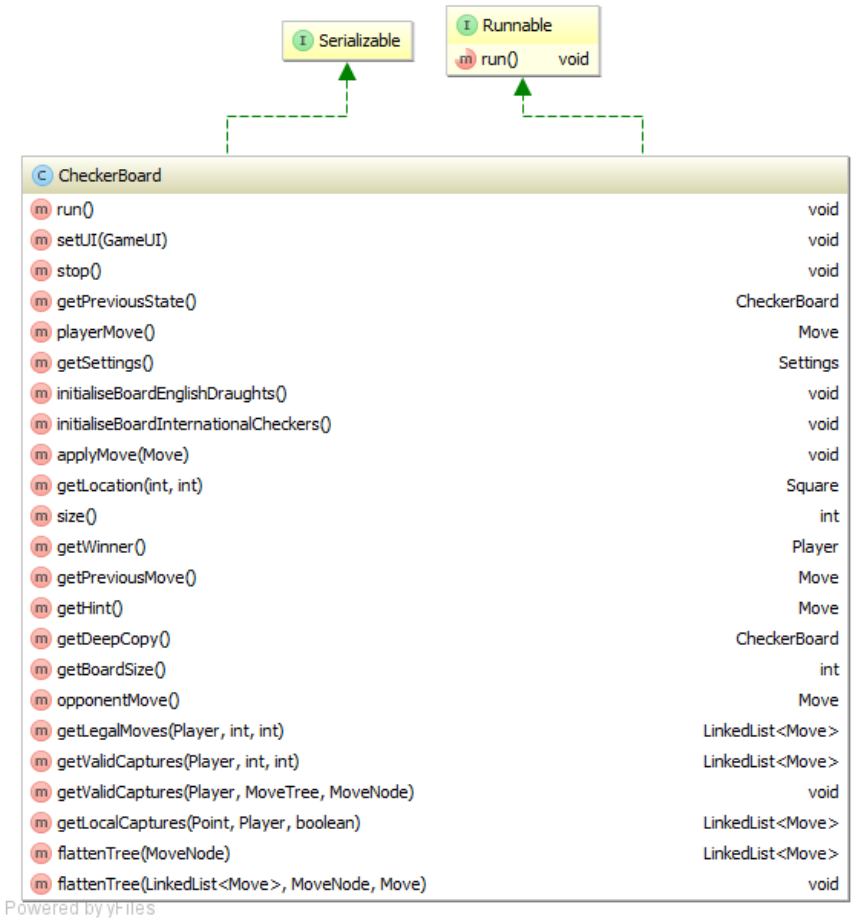


Fig. 3.2 CheckerBoard class UML Diagram

3.2.2 Generating Legal Moves. Calculation of the legal moves was performed in two stages: Standard moves i.e. without any takes were calculated, then any possible takes. The first part was trivial, a check was made to see if a square as diagonal coordinates to the piece in question returned empty, it was added to a move list. In the case of Kings, reverse moves were also checked and added to the list. Takes were a little more complicated as multiple paths could be taken depending upon the rule set; causing pieces to be able to take one or many pieces, or takes with diverging paths as shown in Figure 3.3.

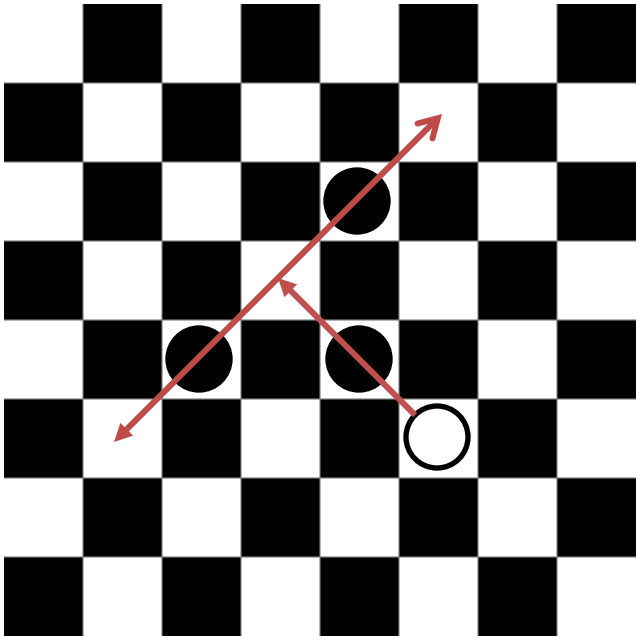


Fig 3.3 Diverging Takes

To account for this problem, the algorithm was designed as a recursive algorithm with would look for local moves only i.e. one step takes from a given position, and construct a tree of moves which could then be condensed into moves by creating a move for every path in the tree. For example the state shown in Figure 3.3 would create a tree structure shown in Figure 3.4, which would flatten to create three possible moves depending upon the rule set as shown in 2.1.

$$\{\{(5,5) \rightarrow (3,3)\}, \{(5,5) \rightarrow (3,3) \rightarrow (1,5)\}, \{(5,5) \rightarrow (3,3) \rightarrow (5,1)\}\} \quad (2.1)$$

Positions were cached to prevent instances whereby cyclical moves could occur.

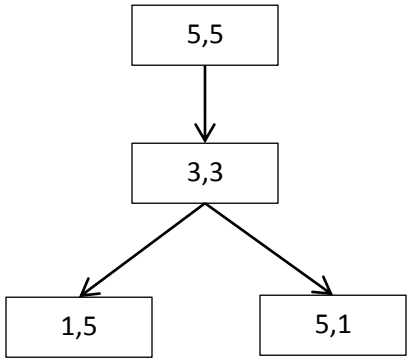


Fig 3.4 Sample tree structure for takes

3.2.3 Settings. The settings class stores all information which is required to persist between uses of the application. This is achieved through serialisation, and being writing to a *dsg* (Draughts Save Game) file. Settings stores colour information, win/loss statistics, difficulty of the AI, the rule set in use, and the

current status of the game after every move so as to prevent data loss in the event of unexpected closure.

3.3 Graphical User Interface

The GUI of the program has two primary functions; handling user input and displaying the current state of the game to the user. Both of these functions are largely handled by a class called GameUI. The first of these two functions is somewhat simpler to explain, but should be easier to understand with knowledge of how the program displays the game on the screen. With this in mind, how the program draws the game on the screen will be described first.

3.3.1 Drawing. Drawing the game is, as previously stated, handled by a class called GameUI. This class extends the Java class JPanel, and all the drawing code is contained in a method called paintComponent, which is a method from the JPanel class which has been overridden. JPanel is layout component which is part of the Java Swing package. The paintComponent method is used by Swing when drawing to the screen, and any calls to Swing's repaint method will invoke the paintComponent methods of any component present in the application window.

When the repaint method is called, it generates a Java Graphics object which is then passed on to any Swing components as an argument to the paintComponent method. The Graphics object contains methods to facilitate the drawing of graphical primitives such as lines along with simple shapes like rectangles. These methods are used throughout the drawing code for the game.

The core of the drawing code is two nested for loops, which iterate over every square in the software's internal board representation. The number of squares on the board could vary with the specific rule set in use at the time for the game, so the code references a variable in the core logic section to ensure the correct number of iterations is carried out.

Before any of the squares are drawn however, the program will determine how large the squares should be drawn. The program scales the size of the squares on the grid to the current size of the program window. Because the game board will always be a square, the board, and therefore the squares in the board will scale to whatever is currently the smallest of either the window's width or the window's height. A final consideration for the scaling is that there is fifty pixels of padding around each side of the board.

Pseudo-code for the scaling calculations follows:

```
windowSize = min(windowWidth, windowHeight);
windowSize = windowSize – 100;~
squareSize = windowSize / numberOfSquaresPerSide;
```

This means that for a window 600 pixels across and 500 pixels down with a board of eight squares will generate a board with squares of size:

(min(600, 500) – 100) / 8 = 50 pixels

Shrinking the window by 200 pixels in the horizontal axis will lead to a board with squares of size:

(min(400, 500) – 100) / 8 = 37 pixels

Note that this result is due to integer division. Integer division is necessary because Java's Swing drawing code takes integers as arguments in most places. Once the size of the squares has been calculated, the squares can then be drawn. For each square on the board, several operations are carried out in a specific order. Note that the vertical size of the squares is multiplied by 0.8 before being drawn, to create an effect of viewing the board from angle, rather than directly above the board.

First, the colour which will be used to draw the square must be determined. While this may sound trivial at first, the program can draw the squares in several different colours to communicate various pieces of information to the user. For example, the start and end squares of a move which the AI has just taken will be highlighted in red and orange, respectively. A full list of the possible colours for the squares follows, in ascending order of importance. (More important information overrides information with a lower priority, as only one colour can be displayed.)

- The lowest priority is simply to draw a white or black square(or whatever the user has selected as colours for the squares)
- The next in the list is highlighting the last move the AI executed. This is done in red for the origin point of the move, and orange for the destination square and any intermediate squares for multi part capture moves
- After this, the next priority is to draw the hints to the user. This is done in yellow
- The next highest priority is highlighting the currently selected piece. This is done with a cyan colour
- Finally, the possible moves for a selected piece should be highlighted. This is done in green

While this priority system is unimportant for many cases, there are some scenarios where it is required. For example, it is possible for the previous move made by the AI to overlap with a square the user would wish to move a piece to. In this case, the priority is needed because the user will care more about possible moves for their pieces, rather than the move the AI made previously.

Once the size of the squares has been determined, and the desired colour of the square has been decided, the squares are drawn individually. The drawing order for the squares from the top left, left to right, one row at a time from top to bottom. The squares are drawn as a filled square in the previously decided colour, and then an outline of an empty square is drawn over the top. For each square, once the square itself has been drawn, there is a check to see if any pieces are present on that square. If there are, the piece drawing code is then used to draw the piece that was found. There is separate code for drawing standard pieces and kings, and this code is used for pieces from either side. To draw pieces from different sides, different colours are passed as arguments to the drawing code.

Drawing pieces is the most complex of the drawing operations, and takes several steps to complete. The first step is the calculation of the four corners of the piece. The size of the pieces is scaled along with the size of the squares, and the position of the piece on the board also has to be taken into consideration.

The first corner to be calculated is the bottom left corner of the piece. The calculation is as follows:

*bottomLeftX = x * squareSize + offset.x + squareSize * 0.1;*
*bottomLeftY = y * squareSize * 0.8 + offset.y + squareSize * 0.55;*

The variables which have not been explained are x, y and offset. x and y are the position of the piece on the board in terms of squares from the top left corner. Offset is a 2D point which holds the offset of the board from the top left corner of the application window. This is currently 50 pixels in both directions, but a variable was created to allow this to be changed if needed. squareSize is multiplied by 0.8 in the vertical direction to fit with the slight vertical shrinking of the squares when they were drawn. The final two constants at the end are used to position the piece inside the square. The output of the calculations based on a tileSize of 60 pixels, and a piece at position 5, 4 are as follows:

*5 * 60 + 50 + 60 * 0.1 = 311 pixels*
*4 * 60 * 0.8 + 50 + 60 * 0.55 = 275 pixels*

Once the bottom left corner position has been chosen, the other three corners are based on the position of the bottom left corner using the size of the piece. The size of the piece is, as already stated, scaled to the square size. The width of the piece is 0.8 of the square width, so the piece will have 0.1 * squareSize on either side of it. The height of the piece is 0.25 * squareSize. Centring the piece vertically is harder, due to the curved edges of the pieces, as well as the 0.8 scaling on the vertical size of the square. This was eventually done using trial and error, rather than using any calculated number.

Once the four corners of the piece have been calculated, the Java class Path2D is used to draw the piece. Path2D is well suited to drawing the pieces because it allows the definition of custom shapes, which can then be drawn or filled with a single call, rather than chaining multiple built-in calls to create the correct shape.

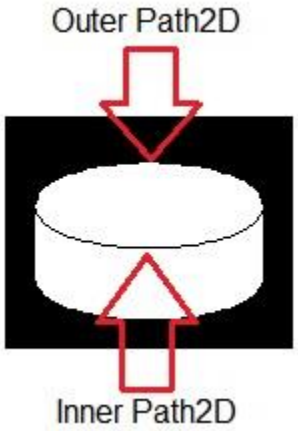


Fig 3.5 Diagram showing which Path2D objects are used to draw which part of the piece

As shown in Fig 3.5 the standard (non-king) pieces are drawn using two Path2D objects, one to draw the outline of the piece and one to draw the internal curve of the piece. This cannot be done with a single

Path2D object, because the with the inner curve added to the outline the Path2D no longer forms a single coherent outline, and this causes some problems when trying to fill in the shape. The king is very similar to the standard piece, but is a little taller and has a second internal curve to create an appearance of two pieces stacked on top of each other.

The four outer edges of the piece are drawn in clockwise order, starting with the bottom curve. While the straight lines are simple to draw, the curved lines are drawn using Bézier curves. Bézier curves are commonly used in computer graphics, and are curves drawn based on four points (known as control points) [6]. Two of these points appear on the curve itself, and are the start and end points of the curve. The two intermediate control points are not present on the curve, and are used to determine the angle of the curve. Regular curves were needed in this case, so the two intermediate points are placed the same distance directly below the start and end points., as shown in Figure 3.6.

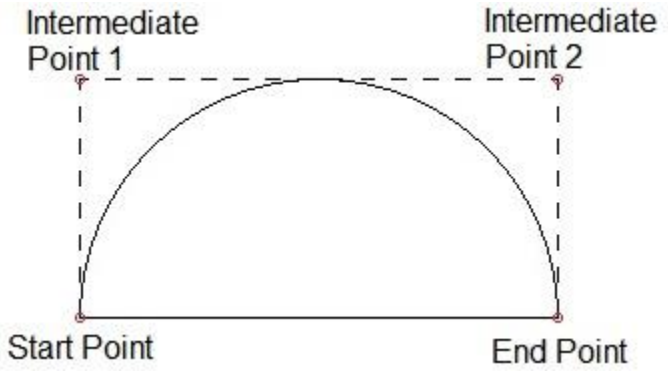


Fig 3.6 Diagram showing how the four points were used to create a curve

3.3.2 User Input. The game is played entirely with the mouse, and all the input in the software is handled by a MouseListener object. This listener gets the X and Y position pixel that was clicked on the screen, and passes this information onto the GameUI class. The GameUI then processes this input in several stages.

The first stage is taking the input position and calculating which square the user intended to click on, if any. This stage can be considered to be somewhat of a mirror of the drawing squares code, as it is essentially the same calculation done in reverse. The calculation takes the offset variable, subtracts it from the input then divides the remainder by the current square size. As with the drawing code, squareSize is a variable that changes based on the current size of the game window. The full calculation is as follows:

```
clickedSquareX = (input.x – offset.x) / squareSize;
clickedSquareY = (input.y – offset.y) / squareSize;
```

Note that the division in this calculation is integer division, so the result will be rounded down to the nearest integer.

Once a candidate square has been calculated, a bounds check is needed to see if the square lies inside the board. This can be considered a check to see if the user clicked an area that is inside or outside the board. The check is very simple, as checks if the candidate X and Y numbers lie between 0 (inclusive) and the size of the board (exclusive). The code for the check is as follows:

If (xTile >= 0 && xTile < board.size() && yTile >= 0 && yTile < board.size())

Where xTile and yTile are the calculated values for the square position, and board.size() gets the size of the board in squares from the core logic. If this check determines that the square is on the board (and therefore that the user clicked somewhere on the board) then the next stage is to determine what the user likely intended to do when they clicked on the board. The main two operations when playing an electronic version of draughts are selecting a piece to move, and then selecting a destination for that piece.

To reflect this, there is an enum in the GameUI class that contains two fields; a “Select” field and a “Move” field. If there is no currently selected piece, the game will be in “Select” mode, and will assume that any clicks on the board will be the user attempting to select one of their pieces. On the other hand, once a piece has been selected the game will be in “Move” mode and interpret any clicks on the board as an attempt to either move the selected piece, unselect the current piece or select a different piece to move.

When the game is in “Select” mode, a simple check is made with the core logic section to see if there is a piece belonging to the user present in the square the user clicked on. If there is, that piece is then selected. In any other case (no piece present, piece belongs to opponent), nothing happens and the game continues to listen for input in “Select” mode. Once a piece has been selected, the square it occupies is placed in a variable in the GameUI class. As long as the piece stays selected, this variable will be used to highlight that square in cyan. In addition, the GameUI class will query the core logic, asking for a list of moves the piece can perform. This list of moves will be held in the GameUI class, and the destination squares for these moves will be highlighted in green. Initially, the program would highlight intermediate steps for moves as well, but this proved redundant due to the fact that an intermediate step will also be a destination square in every possible case.

In contrast, when the game is in “Move” mode, the square that was clicked on is compared with the list of moves that was generated when the current selected piece was clicked on. If the square matches the destination point of any of those moves, the GameUI class then directs the core logic to execute that move. The internal representation of the game will then be updated accordingly, and the current display of the game on the screen will be redrawn to reflect these changes. On the other hand, if the square that as clicked on was not a valid destination, then no move will take place. In either case, the game will return to “Select” mode, the highlight around the piece will be removed and the list of possible moves will be deleted. A third option is that another of the user’s pieces was clicked. In this case, the old piece will be unselected as before, but the new piece will be immediately highlighted and the moves for this piece will be retrieved and displayed on the board. The game will remain in “Move” mode. A flow diagram of the how the system handles input is shown in Figure 3.7.

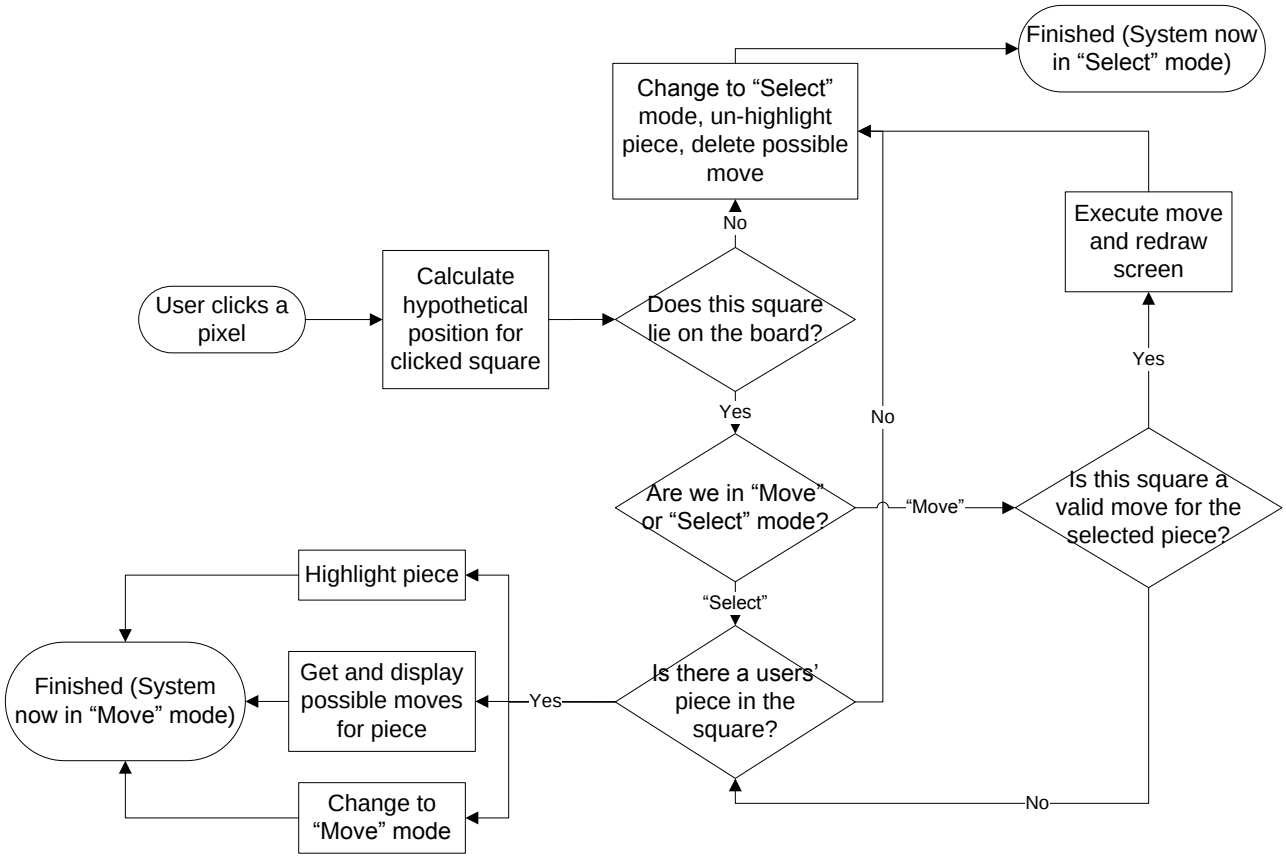


Fig 3.7 Flowchart showing the various stages of processing output

3.3.3 Multithreading. The application was multithreaded to ensure the GUIs correctly handled the updating of events as they occurred. During the first iteration, the GUI would lock once a player had made a move, and would not update the players move nor unlock the thread until the AI had made its counter move, proving a jarring experience, and preventing accessing of menus during that time. Separating the GUI from the rest of the application allowed the GUI to immediately update once notified of a change via the *invokeLater* method in the SwingUtilities class, allowing the user to view their move once made, and interact with the GUI in other ways such as accessing the menus before the AI had finished processing which move to make next, proving an especially useful feature for when higher difficulties were selected, whereby the AI opponent may take some time to select a move.

3.3.4 Game Menus. The primary feature of the second iteration was the inclusion of several menus to adjust certain parameters of the game. The menu hierarchy, shown in Figure 3.8 allowed the user to access tertiary features or options. These were implemented using the Swing JMenu system.

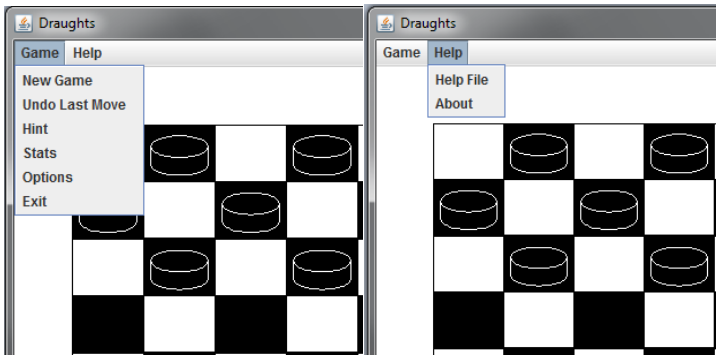


Fig 3.8 Drop-down Menus

The menu options are shown in Table 3.7.

Menu Item	Function
New Game	Reset game board
Undo Last Move	Restore State of board to before player made their previous move
Hint	Display a move the player should take according to the AI
Stats	Show win/lose statistics
Options	Display Options Menu
Exit	Close Application
Help File	Display user manual in PDF form
About	Display About information

Table 3.7 Menu Functions

3.3.4.1 Undo Last Move. Every time the user made a move, a deep copy of the board, using the serialisation method described in Section 3.2.3 was created and stored. The game could be restored to this point upon selection from the Options menu by substituting the stored board with the current one in use and initialising its thread.

3.3.4.2 Options. The options menu, shown in Figure 3.9. The constructor method was made private to ensure that only one instance could be instantiated at any one time.

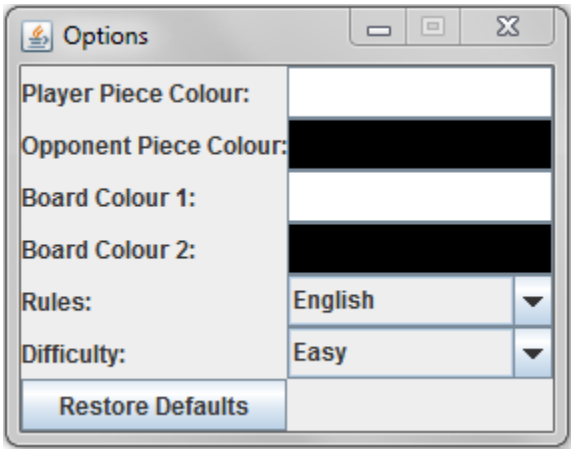


Fig 3.9 Options Menu

A component extended from JButton called ColourPopup was created in order to display the colour of the JColorChooser it opened. This was used to select the piece colour and the board colours. In addition the rules and difficulties could be updated via combo boxes which used the appropriate enums in order to populate their lists. Upon update of any option, the corresponding method in the Settings class was called and saved and the repaint method called in order to instantly update any changes to the game's colour palette. A restore defaults button was also implemented to restore all settings to their predefined values.

3.3.4.3 Statistics Menu. The Statistics Menu, shown in Figure 3.10, displays information regarding the number of wins and losses stored in the settings file.

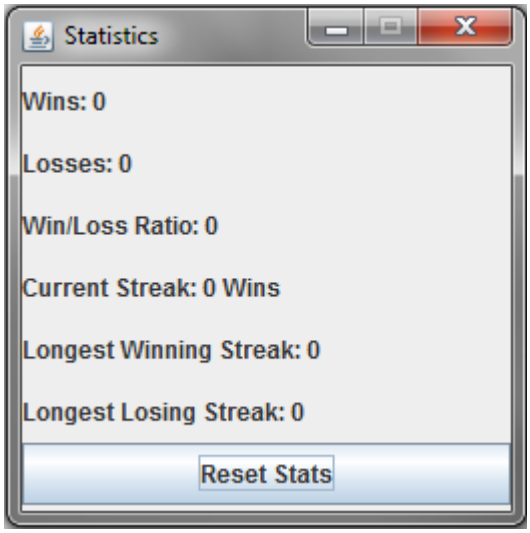


Fig 3.10 Statistics Menu

The number of Wins, Losses, and streaks were stored in the settings file, the win loss ratio is calculated on-the-fly using Equation 3.1.

$$ratio = \begin{cases} {}^{wins}/_{losses} & losses > 0 \\ 0 & otherwise \end{cases} \tag{3.1}$$

Weather the streak is a win or loss is also displayed according to weather the streak is positive (a win streak) or negative (a losing streak). Either way, the absolute value of the streak is displayed. As with the options menu, only a single instance of the statistics menu can be displayed at any one time. The statistics can be reset to zero by pressing the *Reset Stats* button.

3.3.4.4 About Screen. The About screen, shown in Figure 3.11, displays information regarding the program and its creators, the information is hardcoded as it does not change.

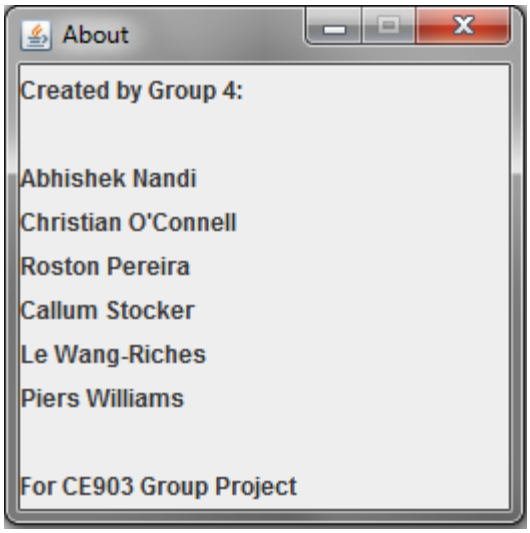


Fig. 3.11 About Screen

3.3.4.5 Help. The help option opens the user guide in *pdf* format.

3.4 AI Opponent

The AI algorithm was implemented using a minimax algorithm [7] due to its suitability to the problem of a two player turn based game. Since the game tree for draughts is so vast that it is not feasible to perform an entire tree traversal, a game tree was constructed for the next few moves, the exact number according to the selected difficulty, and each end move being rated via a heuristic which the AI attempted to maximise. Two main heuristics were implemented: a Naïve form was developed during Iteration 1 which used the number of pieces on the board to form a value of utility for the state of the board. The second was based upon the work of A. Samuel [8] and his work on a rote learning draughts algorithm. Although this method did not employ learning, the system used the concept of a polynomial with several features and an appropriately selected coefficient in order to value the states of the board. Depth of the tree was altered to affect the difficultly, rather than the obfuscation methods mentioned in the Software Requirements Specification [1], which referred to the learning method.

3.4.1 Difficulty. The difficulty of the AI was adjusted via the *ply* to which the game tree was constructed, in effect allowing the AI to look more moves ahead. Three difficultly levels were implemented, although the addition of more would be trivial, for this domain it was decided that three static difficulties were sufficient, labelled “Easy”, “Medium”, and “Hard”. Considered alternatives were to allow the user to manually select how many moves ahead the AI should look, however given that the time complexity of moves increased exponentially relatively quickly, this would introduce an element of redundancy.

The three difficultly levels each corresponded to two, three, and four rounds ahead. Here, a *round* refers to two consecutive moves, one by each player. The selected difficulty from the options menu is stored in the Settings class, which can then be extracted and supplied to the AI method to look ahead. The advantage of this method is that it allows the difficulty to be adjusted during the middle of play, preventing the need to restart a game if the difficulty is found to be too easy or too challenging.

3.4.2 Alpha-Beta Pruning. Minimax algorithm is a move selection algorithm in games whereby each player is attempting to maximise their own utility (or minimise potential future loss) and minimise their opponents. It is ideal for a game such as Draughts as it is a two player game where each play takes turns with a simple win condition which can be described in computational terms i.e. no more pieces or no legal moves. Alpha-Beta pruning [9] refers to a tree search algorithm used in minimax algorithm with minimises the number of evaluated nodes by not evaluating nodes which cannot affect the outcome based upon previously evaluated nodes. The alpha value refers to the maximum value the maximising player will obtain at present and the beta the minimum value the minimising player will obtain if both players play rationally. It adjusts this window to determine if values are outside of this range and therefore can be discounted, as shown in Figure 3.12.

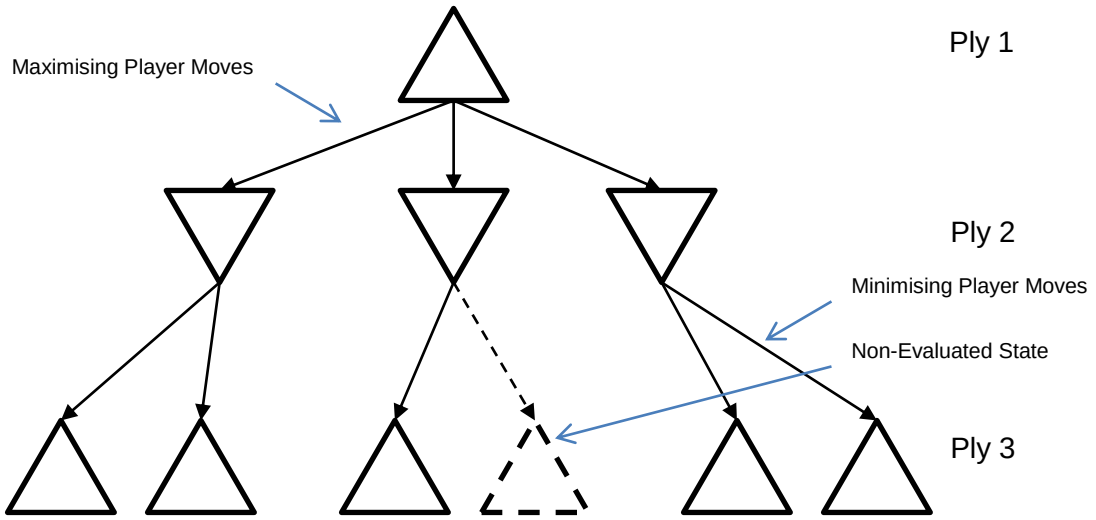


Fig. 3.12 Alpha-Beta Pruning of Game Tree

The Alpha-Beta pruning algorithm allowed the game to look more rounds ahead for a reasonable time cost by preventing the need to expand or evaluate nodes which would not affect the decision of a rational player. In the first iteration, this was implemented as a two stage process: first the move tree was built from the current board state using the relevant method in the core logic. This would include every possible move for the next few moves as determined by the ply level set by the difficulty. This was returned to the AI which would then evaluate the leaf nodes according to alpha beta search and propagate their values upwards to form a decision.

This was optimised in the second iteration: It was noted that a considerable amount of computational work was focused upon forming the actual moves themselves i.e. determining legal moves for a piece, even if that move would later not be evaluated. Therefore the construction of the tree and the alpha-beta pruning search were integrated, such that leaf nodes were evaluated immediately after creation. This allowed the alpha-beta algorithm to prevent the tree formation of irrelevant nodes, cutting down on wasted computation. The code can be found in Appendix A in the file *AIPlayer*, under the method *getBestMoves*.

3.4.3 Architecture of AI. The AI methods were implemented as implementations of an abstract class. Due to the overlap of the methods i.e. using alpha-beta search, these methods could be shared, thus their implementation was defined at a higher level, with their heuristic functions implemented in the implementations of the superclass.

The features were implemented using an interface with a single method which took a board and the maximising player and arguments and returned their value. Each was implemented din their own file rather than as an inner class of the Samuel implementation was there was crossover between the Samuels algorithm and the Naïve implementation, which both used the "DIFF" feature, allowing the refactored code to be used more efficiently, as shown in Figure 3.13 UML diagram.

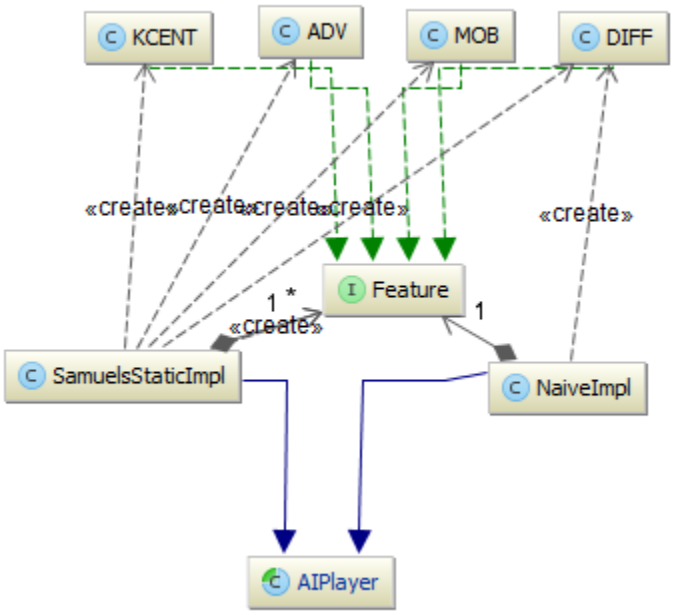


Fig. 3.13 AI Architecture
Generated using JetBrains IntelliJ

3.4.4 Heuristic.

3.4.4.1 Naïve Implementation. The Naïve form of the heuristic was implemented during the first development iteration. Its quick development time allowed other elements such as the alpha-beta pruning and tree construction algorithms to be implemented and tested prior to the more complicated Samuels inspired algorithm. Despite reservations regarding its effectiveness, the heuristic allowed the program to best weaker players fairly consistently, and provided a challenge to medium players at higher difficulty levels. The algorithm worked by scoring each piece according to its status as a king or not, and returning the difference between the active players pieces and the passive players pieces. Standard pieces were given a score of two and kings were given a score of three meaning the AI would sacrifice a king for three standard pieces, considering them of equal value. These values were chosen as

there appears to be agreement in literature of these values [8]. Equation 3.2 shows the value of the board.

$$h(B) = \sum_{i=0}^{N} \sum_{j=0}^{N} \left(-1 \cdot \begin{cases} 1 & B_{ij} = passive \\ 0 & otherwise \end{cases} \right) \cdot \begin{cases} 2 & B_{ij} = stadnard \\ 3 & B_{ij} = king \\ 0 & otherwise \end{cases} \quad (3.2)$$

Where B is the board, N is the number of squares in one dimension of the board, and B_{ij} is the piece at position (i, j). The code can be found under Appendix A in file NaiveImpl.

3.4.4.2 Samuels Implementation. The implementation inspired by Samuels was implemented as a polynomial with different features. The features were inspired suing their equivalents in [8]. Their coefficients were set and adjusted through a process of trial and error to see which produced acceptable play. Its improved effectiveness comes from its ability to identify possible benefit further down the move tree which are not evaluated such as the potential for a piece to become kinged. The equation which combines the different features together is demonstrated in Equation 3.3.

$$h(B) = \sum_{f \in \{features\}} f(B) \cdot c_p^f \quad (3.3)$$

Where B is the board, f is a feature, and c_p^f is the coefficient for feature f at phase p. The idea of a game "phase" was used which was also defined by Samuels, whereby the importance of each feature differs throughout the game. Samuels defined the phase by the number of piece on the board; therefore the same was used in this implementation. The game was separated into three phases of equal length for simplicity as the coefficients were selected manually. Samuels himself initially used three phases but increased this to size later on for this rote learning algorithm. Table 3.8 summarises the features and their coefficients.

Feature	**Purpose**	**Coefficient**		
		Phase 1	**Phase 2**	**Phase 3**
DIFF	This is simply the values returned by the naïve implementation	0.8	0.8	0.95
KCENT	The control of the kings upon the centre of the board	0.2	0.85	0.4
MOB	The mobility of pieces upon the board.	0.2	0.2	0.7
ADV	The number of pieces on pieces guarding the kings row	0.4	0.3	0.1

Table 3.8 Features for Samuel Algorithm.

3.4.5 Randomising Equal Moves. During testing it was noted that, in particular for moves early in the game, the AI would consistently choose the same moves over and over again. The reason was many moves resulted in identical utility from the heuristic function, thus the AI would simply select the first encountered move which returned that utility, as no other method returned a greater value. Whilst not strictly incorrect, this made for dull repetitive gameplay. To circumvent this, all moves which were

evaluated to have the highest equal payoff were stored in list, and then once the algorithm had completed, one would be returned at random introducing an unknown element to the gameplay, which did not affect the AIs ability.

3.4.6 Hint System.

A feature of the system was that the program should give hints to the player as to what should be their next move. The AI system was repurposed for this, instead 'playing' on behalf of the user. The returned move was returned and displayed as a suggestion, which the user could decide to take or not as shown in Figure 3.14.

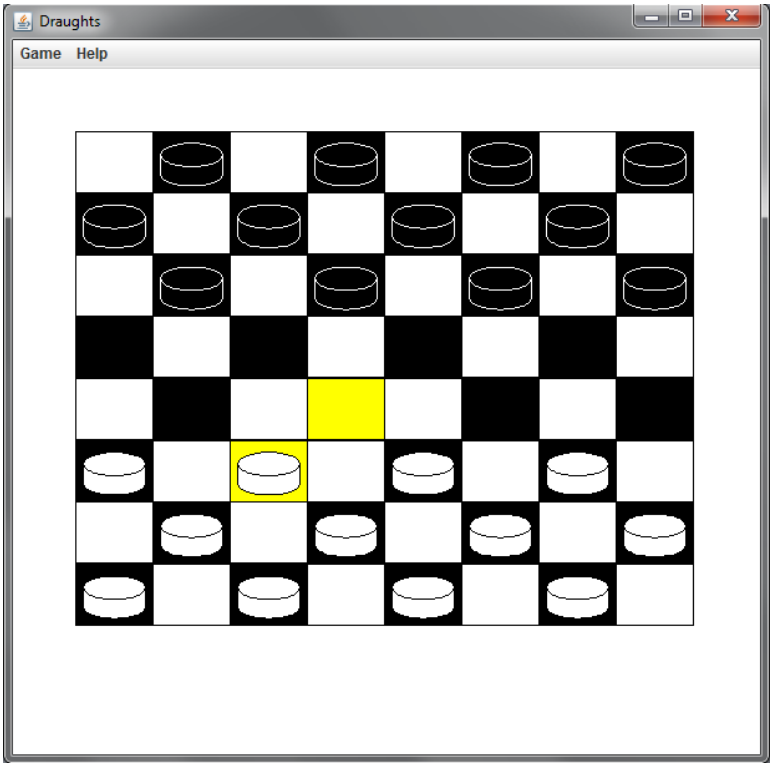


Fig. 3.14 Displaying a returned Hint by the AI

To achieve this, a parameter was included in the tree search method which noted which player the algorithm should be maximising. This allowed the player to be adjusted according to weather the method was called by the main loop or the GUI's Hint option.

4.0 Testing

The following section details some examples of tests used during development.

4.1 Core Logic

The core logic section of the application has gone through a testing phase where a number of tests are simulated on the product to confirm the operational functionality of the product. This testing phase is done so as to pick out bugs in the existing system and to de-bug any bugs that are found as well as to evaluate the performance of the program while a game is played. The testing phase also helps to inform the stakeholder(s) of the quality of the product, its capabilities of the product, the risks involved and allows an overall appreciation of the product.

A software testing consists of a number of extensive and complex steps that are run and re-run till successful results are obtained. Two of the most important kinds of testing are the black box testing and the white box testing. The core logic of the product has undergone both the black box testing phase as well as a white box testing phase. The black box testing of the system will require no prior knowledge of the software code, while the white box testing will require knowledge of the software coding and will test the performance of the software relative to the coding. This section discusses the testing for the board logic testing.

4.1.1 Black Box Testing. The Black Box Testing of the board logic for the game tests to see if the movement of the board pieces are possible or not after the GUI for the board has been designed and set. The tester only needs to know the rules of the game, and needs no knowledge of the source code used to design the game.

The black box testing ignores the internal structure and coding that was used to build the game and focuses on how the game is played and whether it adheres to the rules that have been set for the game. This testing aims to find faults in the gameplay and if faults are found then the test has failed. In that case, the code needs to be restructured and the faults corrected before the test is run again. This process is carried on over and over again till the game runs without errors, examples of this are shown in Figures 4.1 and 4.2.

Test Name	Black Box Test1
Pre-Conditions:	The jar file launches the game interface and the game is playable.
Test	To check the game allows all valid moves to be executed, to check that capturing opponent pieces is possible, to check the evolution of the standard man piece to the king piece once the standard man reaches the last row, and to check that a game session is concluded by identifying a definite winner.
Tested output	There are no noticeable bugs and the game is fully playable, the winner is identified based on the English-draught game rules, and all functionalities work as expected and hence the test is successful.

Test Name	Testing Win Condition
Pre-Conditions:	Board set-up with the player and the AI having their pieces set in their initial positions.
Test	To play the game and identify the winner.
Tested Output	Game is played and black has been identified as the winner, based on game rules and the game session has ended successfully.

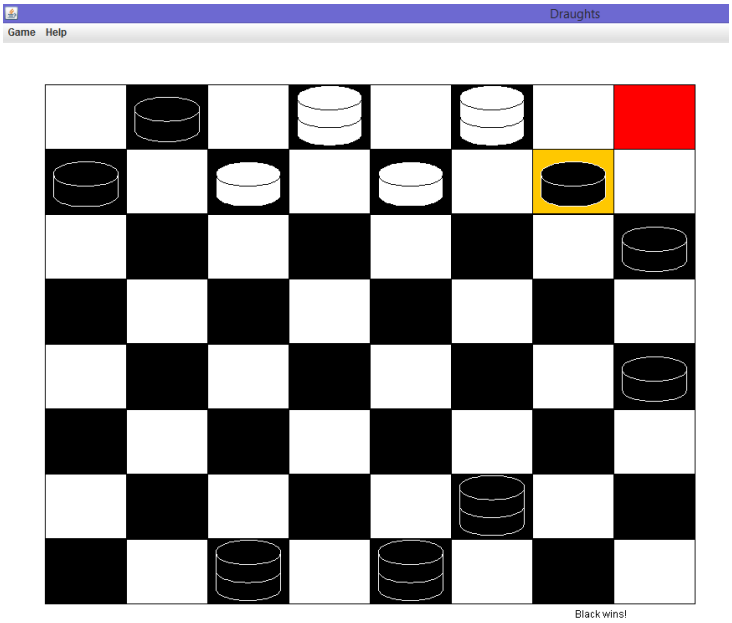


Fig. 4.1 Example of a game being successfully concluded with one side having won according to the game conditions.

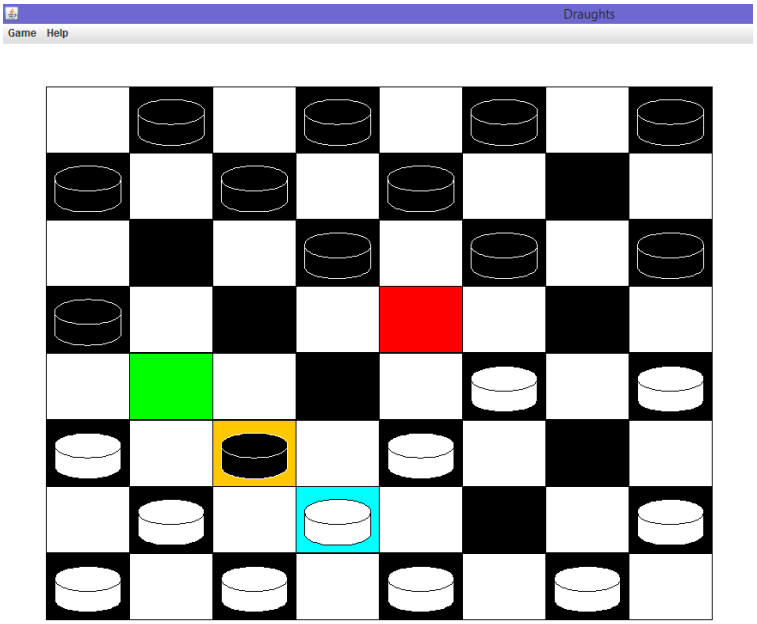


Fig. 4.2 Example of the game showing the last move that the AI has committed, as well as the player piece being selected, as well as allow a legal capture of the opponent AI piece.

4.1.2 White Box Testing. The white box testing for the board logic to check the internal structure and coding that was used to build the game and as well as focuses on how the game is played and whether it adheres to the rules that have been set for the game. This testing aims to find faults in the class structure, method logic, code quality, operational functionality etc and if faults are found then the test has failed. In that case, the code needs to be restructured and the faults corrected before the test is run again. This process is carried on over and over again till the game runs without errors.

In white box testing we should check the overall game is working fine along with each module responding in appropriate manner. The classes should be instantiated as and when required as well as the methods should return appropriate result whenever called. A successful usage of classes including their methods will result in correct move in a particular game. If all moves are correctly executed without any ambiguity then the game will be able to define the winner for a particular game.

Test Name	White Box Test1
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	To check the functioning of the code and test the validity of the classes and methods used.
Tested Output	All classes are valid and all methods are fully functional if called. All classes are able to get instantiated whenever necessary and all methods are producing results as per the internal logic of each method and hence the test is successful.

Test Name	White Box Test2
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are

	understood by the tester.
Test	To check each of the code and test if the classes and methods are written as per the coding standard for java.
Tested Output	All classes and methods are written as per the coding standards for java. All classes used standard method structure and have identifiers as per the standard code structure for java and hence the test is successful.

Test Name	White Box Test3
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	All code logic for each and every move has to be checked to see if any ambiguous move is present which might lead to halt the system.
Tested Output	All move options are checked and verified that they are unambiguous and hence the test was successful.

Test Name	White Box Test4
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	All code has to be checked if there is any infinite logic such as infinite loop.
Tested Output	All logic structures or methods of any class in the core logic that used loops, has been tested and no infinite logic has been encountered and hence the test was successful.

Test Name	White Box Test5
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	To test all classes that are not using main method are serializable.
Tested Output	All classes are checked and those who are not using main method are inherited the properties of serializable class and hence the test was successful.

Test Name	White Box Test6
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	To test all enumeration data types that are used in the core logic has the correct value set.
Tested Output	The value set for all the enumeration data types like pieces/players has been verified and found that they have the correct value set needed for this game play. Hence, the test was successful.
Test Name	White Box Test7
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	To test rule set to determine if all rules are added or is there any missing rule for the game that needs to be added.

Tested Output	The rule set used in the code for the game play had included all necessary rules needed to guide and govern each every move of the game. Hence, the test was successful.

Test Name	White Box Test8
Pre-Conditions:	The code of the core logic is accessible and the code and the functions are understood by the tester.
Test	To test if all object streams that has been opened during execution has been closed properly as well as if it is not closed serializable objects are not saved.
Tested Output	All codes are verified where object or file input/output steams are used and found that all streams were closed once done and hence, the test was successful.

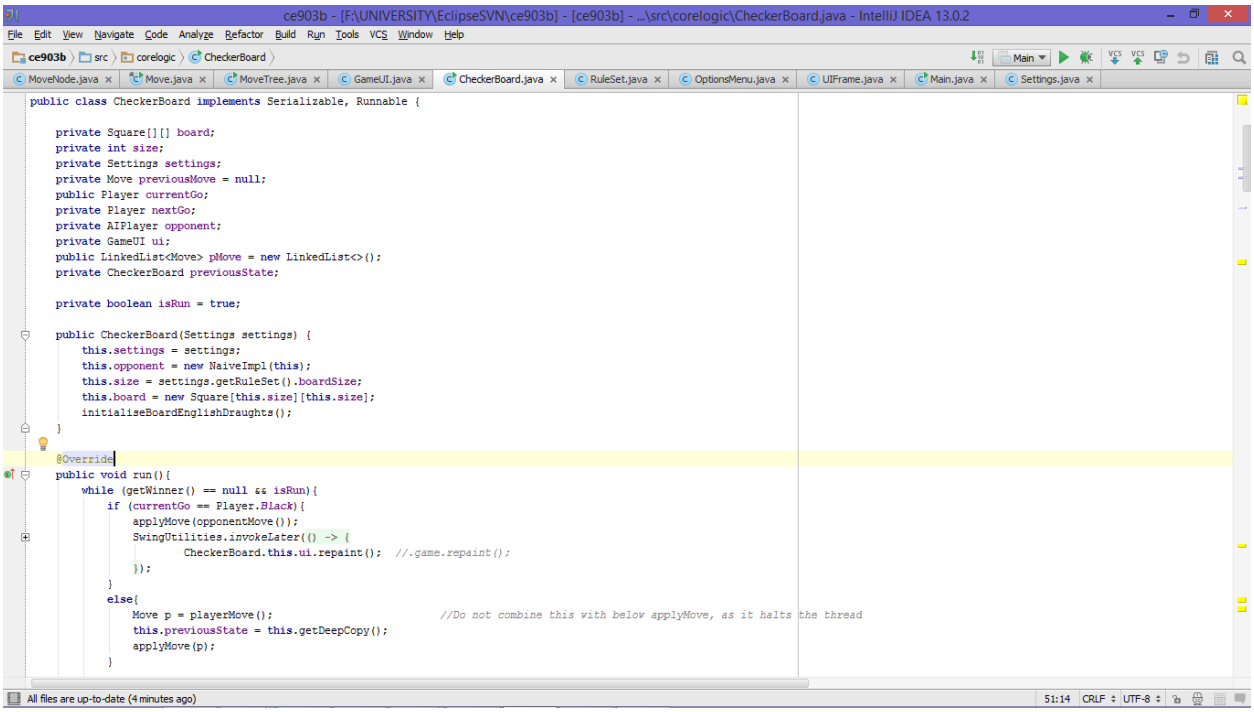


Fig. 4.3 Screenshot showing an example of the methods and classes used to construct part of core logic for the checker board.

4.2 Graphical User Interface

4.2.1 Black Box Testing.

Testing the function of the GUI was performed empirically over a number of observations. In cases where behaviour was unacceptable, parameters were tweaked until the output on the screen was acceptable according to the original design specification.

Path testing was could be carried out on all the stages following the square calculations. (From the Move/Select decision on the flowchart onwards) All the possible paths for the program to follow where

tested in several different scenarios, to ensure there were no problems regardless of the state of the board. To facilitate this, testing was done without the AI playing, so the player had control over both sets of pieces. This testing revealed that there were some states that were not correctly covered, and the program could become locked in a state where the user had not completed a move, but the program would no longer accept any input. To avoid this, a “clean-up” method was created, to ensure the program would always be in valid state no matter what the user did.

4.2.2 White Box Testing.

The input was tested separately. The first stage to be tested was detecting which square had been clicked. This was tested by generating a number (between ten and twenty) of random positions in the window, and having the program print out which squares it calculated contained each of these points. Each of these points, along with the output generated as a result of the points, was labelled to help distinguish between them. These results could then be compared by eye to determine whether they were correct or not. The aim of this test was to robustly test the calculations with an unbiased set of inputs, as a human testing the system would tend to click only on the board itself, rather than the edges, meaning that certain areas of the code would be untested. Humans also tended to click towards the centre of the squares, meaning that the accuracy of the calculations would go untested. This test revealed that the 0.8 scaling on the input had been done incorrectly, and under some circumstances the program would detect input on an incorrect row. The problem lay in the order of operations for the calculation, and once identified was quickly fixed.

4.3 AI Player

4.3.1 Black Box Testing. Black box testing was carried out throughout the development of the application, with many games of Draughts played by the development team. By doing so, it increased the probability of identifying bugs which had not appeared during white box testing. Because of the vast number of possible move combinations, it was infeasible to write tests for every possible combination. Of course this method was not sufficient on its own and was combined with white box testing (Section 4.3.2), as it would often appear that the AI had made a poor move when in fact its true intentions were not clear to the player. This strategy of testing was undertaken for both implementations of the AI. This was of particular interest to the Samuels AI implementation, whereby its effectiveness was evaluated through qualitative means over a number of plays, and its coefficients adjusted accordingly.

4.3.2 White Box Testing. White Box tests occurred during development to ensure that the correct behaviour options were being selected. To achieve this, possible moves and their associated utilities were printed to the console, along with the selected picked move.

Test Name	AI White Box Test 1
Pre-Conditions:	A game against the AI is being played.
Test	Possible move selections and payoffs are printed to the console.
Tested Output	Selected move is part of the highest payoff set
Post-Conditions:	N/A

4.4 Integration Testing

Throughout the development cycle, each component relied upon the other section to provide methods and functionalities. Integration was performed as early as possible during iteration one to mitigate errors occurring due to incompatible interfaces between sections. In keeping with Extreme Programming philosophy, bugs occurring at this stage were fixed immediately by whoever discovered them to prevent them from halting the project at a later date and requiring major refactoring of the code.

4.5 Acceptance Testing

The acceptance testing concluded the project by confirming that the project conformed to the client specification and concluded the user story. This will take place in a demonstration with stakeholders Dr Paul Scott and Dr John Gan.

5.0 Project Management

The management of Group 4 was based on the Extreme Programming methodology due to the members past experiences. Management was conducted through at least one meeting a week whereby workload and deadlines could be discussed. The team changed Team Leader and Secretary changed on rotation every two weeks, allowing nearly all team members have experience with being the secretary, and all but one to be the group leader. All meetings were minute by the secretary that week, although efforts were made to ensure a consistency in style over the weeks to improve readability for all members.

The meeting are the most import aspect of the project management, a number of practices were employed when organising meetings. Meetings were arranged regularly and as often as needed, especially before deadlines. Periods between deadlines we would still ensure a meeting per week at the very least. The meeting are arranged when all members are free, during the meeting we ensure all members are aware of work done and task set for each other. However naturally there were times when not all team members could be present due to other commitments, so during those meetings, a present team member would volunteer to update the absent team member. Being aware of progress meant the team was more easily monitor the progress overall. Thus work not completed are assigned extra time and if it was completed the member can be assigned a new task.

All members were expected to contribute to meetings regardless of their status as leader or otherwise. Members were encouraged to volunteer for tasks, allowing them to decide their own level of involvement. This approach was taken to allow members to gravitate toward areas where they felt their talents would be best used. However in practice this did results in some team members contributing to significantly less to the project than others.

Conflict arose around communication issues outside the team meetings, such as clarification regarding code choices or implementations not being pursued, causing problems during integration of the main components. This also had a much larger effect with regards to the second iteration, where only one team member contributed to the code, the remainder believing it was the responsibility of someone else to perform assigned tasks. This significantly held the project up and had a knock on effect to the quality of work later in the project which became more rushed due to the approaching deadlines, and resulted in some features not being implemented in the final product.

To conclude Team 4 project management overall was largely successful with regards to arranging meetings and meeting deadlines. However, there were significant problems with communication. Between meetings some members did not make enough of an effort during the implementation, or else wasted valuable time implementing features which already existed. If the project were to continue, focus would be on resolving the lack of communication. Despite this team 4 was able to finish and complete work on time for the first iteration, and most of the second iteration.

6.0 Conclusions

This document details the implementation and outcomes of the Draughts game project. The task was to implement a basic draughts game with configurable options and an AI opponent. To this extent the project was a success. A number of processes that has been discussed and implemented during the process of development of the game are matters such as user friendly features of the GUI, a logic base on which the game can be played, an AI opponent. The system offers versatility to the user in the way of configuration of the game play; the board pieces as well as the board colours are changeable as per the choice of the user. The system also offers the user to choose the level of difficulty that the user wants to play at. Some changes occurred between the features specified in the Software Requirements Specification and the final implementation due to practical constraints such as changes to the GUI layout, and a learning element of the AI which was not achievable in the specified time. However a static version of Samuel's checkers algorithm was implemented. The game plays at the level on which it is expected to, meeting all necessary gameplay standards for a Draughts game.

Use of the Integrated Development Environment IntelliJ helped facilitate the project due to its integration with many version repositories, in the case of this product the Subversion Revision Control system. In addition, the IDE allowed visualization methods to view progress and architecture of the project as it evolved, to allow quick comparison with designs during development. The subversion revision control allowed changes to be made without the fear of accidental breakage as it could be rolled back if necessary; as well a method of updating what had been done.

There was some shortfall observed however, of most note is the functionality of different rule sets was largely not implemented beyond different board sizes and starting players due to time management issues. Also while implemented, the Samuels AI did not play optimally. Finding the optimal features and coefficients was not permitted by the time remaining; however this could make for an interesting future work, of which there is still a significant scope. Other options for future work include features such as a two player mode and networked play.

To conclude, the implementation of the Draughts game reached an acceptable standard, all necessary features were implemented, however there is still room to add additional features and improve the experience of the player.

7.0 References

[1] Group 4, “Software Requirements Specification”, Ver. 1, Dept. CSEE, Univ. Essex, Feb 5th 2014.

[2] D. Wells, “Extreme Programming: A Gentle Introduction”, Extreme Programming, Oct. 8th 2013, [Mar. 16th 2014], URL: http://www.extremeprogramming.org.

[3] Masters Traditional Games, “The Rules of Draughts”, Internet: http://www.mastersgames.com/rules/draughts-rules.htm [Feb. 3rd, 2014].

[4] “International Draughts”, MindSports, Internet: http://www.mindsports.nl/index.php/arena/draughts/492-international-draughts, [Mar. 17th 2014].

[5] J. Schaeffer, “Checkers Is Solved”, Science, vol. 317, no. 5844, pp. 1518–1522, Sept. 14, 2007.

[6] M. Kamermans *A Primer on Bézier Curves* [Online]. Last Accessed 14/03/2014 Available: http://pomax.github.io/bezierinfo/.

[7] G. Strong, “The Minimax Algorithm”, Trin. College Dublin, Feb. 10, 2011.

[8] A. L. Samuel, “Some Studies in Machine Learning Using the Game of Draughts”, IBM Journal of Research and Development, vol. 3, July 1959, pp. 211 – 229.

[9] D. Edwards & T. Hart, “The Alpha–beta Heuristic (AIM-030)”, Massachusetts Institute of Technology. Dec. 4th, 1961.

8.0 Appendices

Appendix A – User Documentation

Overview

Draughts is single player game against an AI opponent with varied levels of difficulty that can be set before the start of the game.

System Requirements

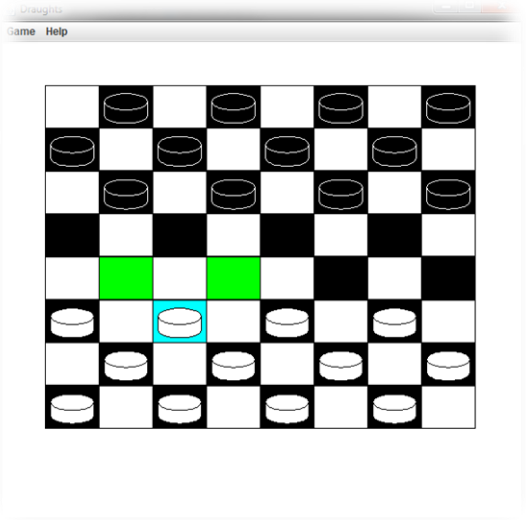
- A 2GHz Dual Core or similar processor.
- 2GB RAM
- Mouse
- Keyboard
- Includes Memory Storage
- Java Virtual Machine (J.V.M).
- 10MB Hard Drive Space

Installation

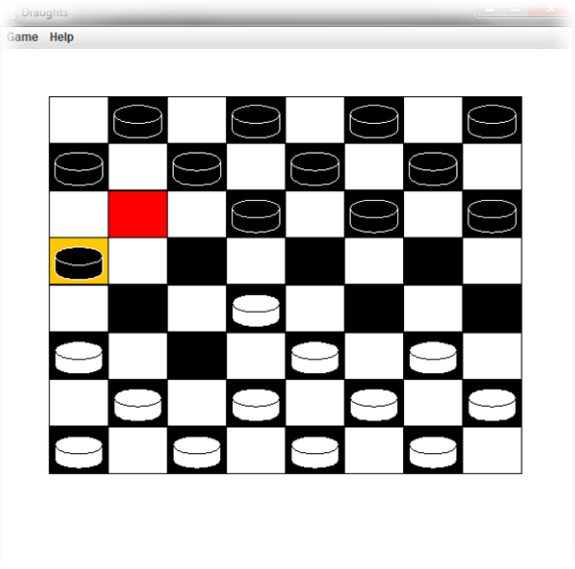
1. Install the latest version of the Java Runtime environment from: http://www.oracle.com/technetwork/java/javase/downloads/java-se-jre-7-download-432155.html
1. Run the Draughts.jar file

Gameplay

The player takes it in turns with the AI to make moves. Pieces may move in the direction of play diagonally one square at a time, or jump a piece of the opposite colour to take it. To win, either take all the opponents pieces or leave the opponent with no legal moves. The direction of play for the user is upwards. Selecting a piece will highlight it in blue and its available moves will be highlighted in green, select on to make a move or select another piece to view more moves.



The game piece contained within the orange highlight shows which piece was moved by the AI, and the empty square with the red highlight shows the initial position from which the piece was moved.



If a piece reaches the opposite side of the board, it will be *Kinged*, and is now free to move and capture both backward and forward.

Software Features:

- Being a new Game by going to *Game* > *New Game*
- Change the difficulty of the AI from the options menu under *Game* > *Options*.
- Alter the colour of the pieces and board from the Options menu. Click the colour option next to the desired options and select from the presented options.
- Select *Game* > *Hint* to be shown a possible move in yellow.
- Select *Game* > *Undo Last Move* to undo the most recent move.
- The game saves automatically, and can be resumed after closing the application.
- Go to Game > Stats to view the win/loss statistics.

Appendix B – Meeting Minutes

Please See Attached CD-ROM

Appendix C – Code Listings

Please See attached CD-ROM.