Assignment 3

Zichen Jiang - jiangz2 - 1320889
Danish Khan - khand5 - 1217176
Kelvin Lin - linkk4 - 1401464
Hasan Siddiqui - siddih8 – 1450148

Course Code:
SFWR ENG 2AA4/COMP SCI 2ME3

Due Date:
April 8th, 2016

By virtue of submitting this document we electronically sign and date that the work being submitted is our own individual work.

Table of Contents

1 Introduction and Architecture

This report will document the design and the decisions made in Assignment 3 for Sfwr Eng 2AA4/Comp Sci 2ME3. The report will begin with an overview of the architecture and modules used in the application. Then the decomposition hierarchy will be examined in order to highlight its dependencies and to prepare a premise for a discussion on anticipated changes and other traditional software engineering practices. This report will end with a test plan, which will show that the application does fulfil the requirements up to the level specified in the document.

The architecture chosen for this application is the Model-View-Controller (MVC) architecture. The MVC architecture was chosen because it accentuates the principle of separation of concerns. The MVC architecture consists of three main components: the model, the controller, and the view. With the MVC architecture, the model represents data related to the logic the user works with, the view represents the user interface, and the controller facilitates the interaction between the model and the view. We believe that by using this architecture, different aspects of the program could be separated in order to facilitate testing and implementing future changes.

Note that this report encompasses the design decisions for assignment 1, assignment 2, and assignment 3. This is deliberately done in order to demonstrate growth, how design decisions from the first assignment impacted the second assignment, how decisions made in the second assignment impacted the third assignment, and how the group handled poor design from the first and second assignment.

2 Modular Decomposition and Hierarchy

The application was designed using a top down approach. The top down approach was used in order to facilitate modular design and to accentuate separation of concerns. By using top down design, the application can be decomposed into modules, which are responsible for a single work assignment. Knowing the individual modules will allow programmers to program using the bottom-up approach, which would allow for early testing, and quick implementation of the application.

The figure below shows the modular decomposition hierarchy for the application. Arrows point towards increasing modularity (from less modular to more modular):

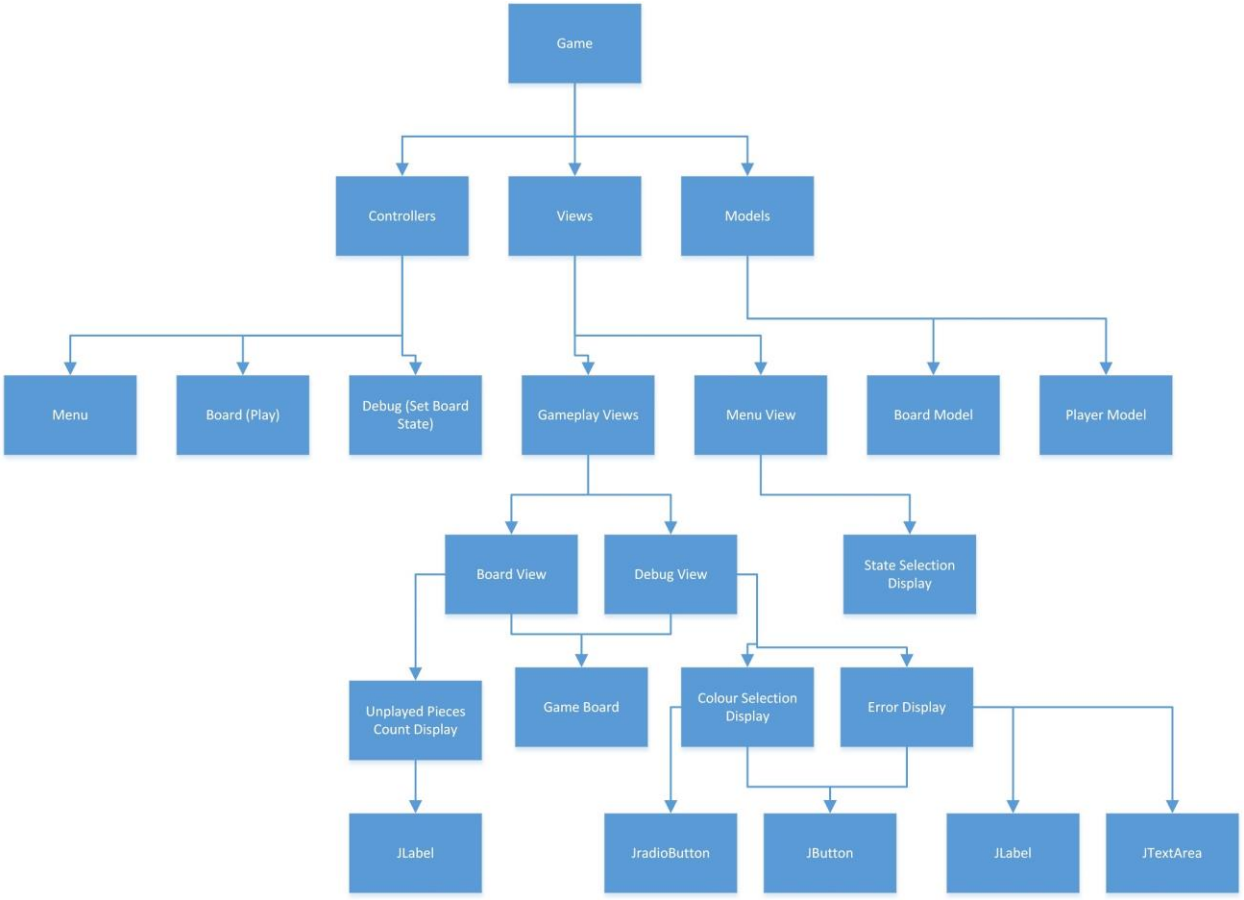

Figure 1: The Decomposition Hierarchy

From the diagram above, it is clear that the game is decomposed into controllers, models, and views. This reflects our design decision to implement Six Men's Morris using the MVC architecture. The controller is then decomposed into the Menu, Board, and Debug modules. The menu is used to fulfill the requirement of having multiple ways to start a game (i.e. by starting a new game, by loading a saved state, and by setting a preset state). The board represents the requirement that the user has to play the game. The debug state represents the requirement that the user should be able to place pieces on the board in order to set the initial state. The view is decomposed into the gameplay views, and the menu view. The menu view fulfills the requirement that the user must be able to see the menu when they are in the menu state. The gameplay view is further decomposed into the board view and the debug view which facilitates displaying the user interface to the user. Finally, the models are decomposed into the two main components of the game: Board and Player. The board facilitates all of the game operations such as checking win conditions, draw conditions, and representing the state of the board. The player model fulfills the requirement of allowing users to place pieces onto the board.

3 Module Guide

3.1 MIS

3.1.1 CLASS: CIRCLE

Defines a mathematical representation of a circle using its center point and radius. Contains access programs to field variables, and to detect user input.

INTERFACE

USES

Point

TYPE

None

ACCESS PROGRAMS

Circle(Point center, double radius)

Constructor method required to create object of type Circle with a radius and center point.

getIntDiameter(): int

Returns the diameter d of the circle as an integer, $d = 2r$

getIntPointX(): int

Returns the x-coordinate of the center point as an integer.

getIntPointY(): int

Returns the y-coordinate of the center point as an integer.

getIntRadius(): int

Returns the radius of the circle as an integer.

isMouseOver(Point mouse): boolean

Returns TRUE if mouse is pointing over the circle, otherwise returns FALSE

3.1.2 CLASS: DEBUGCONTROLLER

Creates the window and all labels or buttons needed to access and update the view which in turn will change values in the model.

INTERFACE

USES

BoardView

DebugController

TYPE

None

ACCESS PROGRAMS

DebugController(int N)

Construct the state based on the number of layers.

3.1.3 CLASS: ERRORDIALOG

Defines a dialog box to display error messages to the user. Contains an access program to respond to the user's input.

INTERFACE

USES

JDialog, ActionListener

TYPE

None

ACCESS PROGRAMS

ErrorDialog(JFrame parent, String title, String message)

Initializes a dialog to display any errors found during the execution of the application

actionPerformed(ActionEvent e)

Responds to the user's input

3.1.4 CLASS: MENUCONTROLLER

Defines a controller to mediate the views and models used in the menu.

INTERFACE

USES

MenuView, JFrame

TYPE

None

ACCESS PROGRAMS

MenuController()

Instantiates the view and any field variables used in the module.

run(): void

Runs any operations associated with the controller.

getJFrame(): JFrame

Returns the JFrame.

3.1.5 CLASS: MENUVIEW

Defines a view for the menu screen.

INTERFACE

USES

Screen

TYPE

None

ACCESS PROGRAMS

MenuView(int N)

Instantiates the objects on the screen and any field variables used in the module.

getState():int

Returns the state of the application.

updateScreen(): void

Updates the screen.

paintComponent(Graphics g): void

Draws the required components onto the screen.

3.1.6 CLASS: PLAYER

Each player is determined by two integers. An integer that represents their color , and the number of pieces they have left to place.

INTERFACE

USES

None

TYPE

None

ACCESS PROGRAMS

Player(int color, int numberOfUnplayedPieces)

Initializes field variables

getColor():int

Returns the color of the player.

getNumberOfUnplayedPieces():int

Returns the number of pieces the player has yet to place.

placePiece(): void

Models the action of the user playing a piece on the board.

3.1.7 CLASS: POINT

Defines a mathematical representation of a circle using its x-coordinate and its y-coordinate.

INTERFACE

USES

None

TYPE

None

ACCESS PROGRAMS

Point(double x, double y)

Initializes the field variables

getX(): double

Returns the x-coordinate.

getY(): double

Returns the y-coordinate.

getIntX(): int

Returns the x-coordinate as an integer.

getIntY(): int

Returns the y-coordinate as an integer.

getDistance(Point that): double

Return the distance between two point objects, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

3.1.8 CLASS: RECTANGLE

Defines a mathematical representation of a rectangle defined by the top left, top right, and bottom left corners.

INTERFACE

USES

Point

TYPE

None

ACCESS PROGRAMS

Rectangle(Point topLeft, Point topRight, Point bottomRight)

Initializes the field variables

getTopLeft(): Point

Returns the top left point of the rectangle.

getTopRight(): Point

Returns the top right point of the rectangle.

getBottomRight(): Point

Returns the bottom right point of the rectangle.

getIntWidth(): int

Returns the width of the rectangle, $w = | x_{topleft} - x_{topright} |$

getIntHeight():int

Return the height of the rectangle, $h = | y_{topright} - y_{bottomright} |$

getTopLeftIntX(): int

Return the x-coordinate of the top left corner as an integer.

getTopLeftIntY(): int

Return the y-coordinate of the top left corner as an integer.

getBottomLeft(): Point

Return the bottom left point, defined by $(x_{topleft}, y_{bottomright})$

3.1.9 CLASS: SCREEN

Declares a function that will be used in classes that extend from it. In this assignment, that would be the classes menuView, and boardView. It is a template for the views.

INTERFACE

USES

None

VARIABLES

None

ACCESS PROGRAMS

updateScreen():void

Updates the screen

3.1.10 CLASS: BOARD

This is an abstract representation of the game board. It keeps the state of each piece in a 1 dimensional array in order to reduce run time and space.

INTERFACE

USES

None

TYPE

None

ACCESS PROGRAMS

Board(int N)

Constructs an array representation of the board.

Board(int N, int[] pieces)

Constructs an array representation of the board given a preset state.

setPieces(int[] pieces): void

Initializes the pieces array.

getN(): int

Returns the number of squares on the board.

setPieceState(int number, int state): void

Set the state of a piece on the board

getBoardState: int[]

Return the current state of the board

getPieceState(int number): int

Return the current state of the piece (black, red or blue)

millExists(int i): int[]

Return the positions of the pieces that forms a mill.

Returns [-1,-1,-1] if no mill found.

onlyMillsLeft(int color): boolean

Return true if there is only one mill left, and false otherwise.

boardIsEqual(int[] board1, int[] board2): boolean

This method is used to check if the given two boards are the same.

3.1.11 CLASS: BOARDCONTROLLER

This is a controller for the board class. It acts as an intermediary between the Board model (Board.java), and the Board view (BoardView.java).

INTERFACE

USES

BoardView

Player

AI

TYPE

None

ACCESS PROGRAMS

BoardController(int N)

Create a new board controller with an N-men's Morris board.

BoardController(int N, int[] boardState)

Create a new board controller with an N-men's Morris board and a specified board state.

BoardController(int N, int[] boardState, int turn, int state)

Create a new board controller with an N-men's Morris board, a specified board state and a specified player's turn.

3.1.12 CLASS: BOARDVIEW

This class displays the board to the user.

INTERFACE

USES

Screen

Board

Circle[]

TYPE

None

ACCESS PROGRAMS

BoardView(int N)

Constructs the screen needed to play the game, and adds all EventListeners needed to obtain input from the user.

BoardView(int N, int[] boardState)

Construct the screen needed to play the game given a certain state, and adds all EventListeners needed to obtain input from the user.

BoardController(int N, Boolean ExistsAI)

Constructs the board with the option of adding an AI.

pieceNotTaken(int number): boolean

Return a boolean value that determines if a piece is already placed in a certain location.

initAI(int AI_color): void

Initializes the AI for the game, and assigns the AI a color. If no color is given, then then color assigned is randomly determined.

getBoardStates(): int[]

Return the state of the board (player Red or player Blue).

setBoardState(int number, int state): void

Set the state of the board.

getBoardState(int number): int

Return the state of the board.

getCircles(): Circle[]

Return the array of all circles in the board

setState(int[] states): void

Set the states of the game.

updateScreen(): void

Updates the screen.

paintComponent(Graphics g): void

Draw board.

millsExist(int i): boolean

Returns whether a mill including the piece at index i exists.

existsOnlyMills(int colour): boolean

Returns whether there only exists mills on the board with the specified colour.

checkWinner(): int

Returns whether or not there is a winner in the game: 0 if there is no winner, 1 if the winner is blue, 2 if the winner is red.

getRepeats(): int

Returns the number of repetitions.

3.1.13 CLASS: GAME

Launches the implementation of Six Men's Morris.

INTERFACE

USES

None

TYPE

None

ACCESS PROGRAMS

main(String[] args)

The Main method that creates an object of type controller and sets the window to be visible. Essentially runs the entire program.

3.1.14 CLASS: AI

Provides calculations for second player moves when playing against the computer.

INTERFACE

USES

None

TYPE

None

ACCESS PROGRAMS

AI(BoardView board, int ai, int player)

Constructs the AI class.

updateBoardView(BoardView boardView): void

Updates the view to the most recent view.

nextPlace():int

Finds the next available position to place a piece.

nextRemove(): int

Finds the next available piece belonging to the player to remove if and only if a mill was detected on this AI turn. The AI will remove first a piece from a Player's mill. If no mill was found, then it will remove the Player's next available piece.

nextMove(): int[]

Finds a piece on the board that is available to move, then moves it appropriately.

3.2 MID

3.2.1 CLASS: CIRCLE

Defines a mathematical representation of a circle using its center point and radius. Contains access programs to field variables, and to detect user input.

IMPLEMENTATION

USES

Point

VARIABLES

center: Point

The center point of the circle.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by getIntPointX(), getIntPointY(), and isMouseOver().

radius: double

The radius of the circle.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by getIntDiameter(), getIntRadius(), and isMouseOver().

ACCESS PROGRAMS

Circle(Point center, double radius)

Constructor method required to create object of type Circle with a radius and center point.

getIntDiameter(): int

Returns the diameter d of the circle as an integer, $d = 2r$.

getIntPointX(): int

Return the X - coordinate of the center point

getIntPointY(): int

Return the Y - coordinate of the center point

getIntRadius(): int

Return the radius of the circle as an integer.

isMouseOver(Point mouse): boolean

Returns TRUE if mouse is pointing over the circle, otherwise returns FALSE

Return x > center.getIntX() - radius ^ x < center.getIntX() + radius ^ y > center.getIntY() - radius ^ y < center.getIntY() + radius;

3.2.2 CLASS: DEBUGCONTROLLER

Creates the window and all labels or buttons needed to access and update the view which in turn will change values in the model.

IMPLEMENTATION

USES

JFrame

BoardView

JRadioButton

ButtonGroup

JButton

DebugController

VARIABLES

jFrame: JFrame

A variable that represents the JFrame object.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor.

boardView: BoardView

A variable that represents the BoardView object.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor.

NUMBER_OF_PIECES = 6: int

Instantiate number of pieces per player, if this was 9 men's morris, it would change to 9 etc.

This variable is final.

BLUE_STATE = 1: int

Blue state is index 1.

This variable is final.

RED_STATE = 2: int

Red state is index 2

This variable is final.

FONT_SIZE = 25: int

Set default font size

This variable is final.

DEFAULT_SCREEN_WIDTH = 500: int

Set default screen width

This variable is final.

DEFAULT_SCREEN_HEIGHT = 500: int

Set default screen height

This variable is final.

blue, red, black: JRadioButton

Series of JRadio buttons.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor and MouseClickEventHandler.mousePressed().

buttonGroup: ButtonGroup

Put buttons in a group so only one can be pressed at a time

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor.

playGame: JButton

Will change the application state to regular gameplay

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor and resizeText();

N: int

Number of layers/ rectangles

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor and playGameMouseClicked();

ACCESS PROGRAMS

DebugController(int N)

Construct the state based on the number of layers.

Instantiate debugging window

Font size scalable to window,

*frame width * FONT_SIZE / DEFAULT_SCREEN_WIDTH*

3 JRadio buttons

playGameMouseClicked(MouseEvent e): void

This method performs when the play game button is clicked.

if boardIsLegal is true

Set up boardController

boardIsLegal(): boolean

Return a boolean value that determines if the board is of legal creation by the user.

if error

new error dialog

return false

checkNumbers(): String

This method checks the debugging board to see how many pieces of each colour are present, if the number is illegal, returns error message.

<table>
<tr><td rowspan="6">blueCount <= NUMBER_OF_PIECES</td><td rowspan="5">redCount <= NUMBER_OF_PIECES</td><td rowspan="2">blueCount <= 1</td><td>redCount < 3</td><td>Legal</td></tr>
<tr><td>redCount >= 3</td><td>Illegal</td></tr>
<tr><td rowspan="2">redCount <= 1</td><td>blueCount < 3</td><td>Legal</td></tr>
<tr><td>blueCount >= 3</td><td>Illegal</td></tr>
<tr><td colspan="2">blueCount > 1</td><td>Legal</td></tr>
<tr><td colspan="2">redCount > 1</td><td>Legal</td></tr>
<tr><td colspan="3">redCount > NUMBER_OF_PIECES</td><td>Illegal</td></tr>
<tr><td colspan="4">blueCount > NUMBER_OF_PIECES</td><td>Illegal</td></tr>
</table>

Table 1: checkNumbers() tabular expression

resizeText(): void

Adjusts the font for all text involved, making it able to be resized based on the window size. Sets the font of blue, red, black and playGame.

Equation for font: $FontSize = Width \times Default\_Font\_Size / Default\_Screen\_Width$

updateView(): void

Updates the view if and where it is needed by using invalidate and repaint.

MouseClickEventHandler *implements* MouseListener

mouseClicked(MouseEvent e): void

Called if the mouse is clicked on a board node, and one of the JRadio buttons (red, blue or black) is selected.

Instantiate points
for i in range of length circles
if blue is selected
make circle at i blue
else if red is selected
make circle at i red
else if black is Selected
make circle at i black
update view

3.2.3 CLASS: ERRORDIALOG

Defines a dialog box to display error messages to the user. Contains an access program to respond to the user's input.

IMPLEMENTATION

USES

JDialog
Action Listener
BorderLayout

VARIABLES

FONT_SIZE = 0: int

The default font size.
This variable is final.

ACCESS PROGRAMS

ErrorDialog(JFrame parent, String title, String message)

Catches errors that may occur in the application while the user is running it.
Set the font
Button when clicked analyses board
Instantiate errorMessages

actionPerformed(ActionEvent e)

If an action can be performed, then there is no need to show the error message, so this will set the visible value to false.

3.2.4 CLASS: MENUCONTROLLER

Defines a controller to mediate the views and models used in the menu.

IMPLEMENTATION

USES

JFrame

MenuView

VARIABLES

jFrame: JFrame

This represents the jFrame to be displayed to the user.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by run() and getJFrame().

view: MenuView

This represents the menuView to be displayed to the user.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by run().

nextState: int

This returns the next state to the user.

This represents the jFrame to be displayed to the user. This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by getNextState().

ACCESS PROGRAMS

MenuController()

Method to construct the output that the controller will handle everything inside of it. It instantiates views.

run(): void

Update the screen whenever the components need to be resized.

getJFrame(): JFrame

Return the window object.

getNextState(): int

Return the next state of the application.

Play Game Button Pressed		Go to BoardController
Play Game Button Not Pressed	Debug Button Pressed	Go to DebugController
	Debug Button Not Pressed	Do Nothing

Table 2: getNextState() tabular expression

3.2.5 CLASS: MENUVIEW

Defines a view for the menu screen. Instantiates what thing must be communicated to the user when called by the controller.

IMPLEMENTATION

USES

Screen

JLabel

JButton

VARIABLES

title: JLabel

The label title. This represents the jFrame to be displayed to the user.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by draw(), updateScreen(), and paintComponent().

playGame: JButton

User will click to go directly to play state.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by playGameMouseClicked(), draw(), updateScreen(), and paintComponent().

debug: JButton

User will click to go to debugging and then to play state.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by MenuView(), debugMouseClicked(), draw(), updateScreen(), and paintComponent().

state: int

Keeps track of the state the game is in.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by MenuView(), and getState().

boardState: int[]

Keeps track of the state of the saved board.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by loadGameMouseClicked().

turn: int

Keeps track of the turn of the saved board.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by loadGameMouseClicked().

gameState: int

Keeps track of the game state of the saved board.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by loadGameMouseClicked().

removePiece: boolean

Keeps track of whether the player is eligible to mill.

This variable is kept private to reduce coupling between modules, and implement

information hiding. It is maintained by loadGameMouseClicked().

defaultFontSize: int

Sets the default font size.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by draw(), updateScreen(), paintComponent().

defaultScreenWidth: int

Sets the default screen width.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by draw(), updateScreen(), paintComponent().

N: int

The number of layers.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by MenuView(), draw(), updateScreen(), and paintComponent().

playGameAI: JButton

Allows the player to initialize an AI enabled game.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by MenuView(), playGameAIMouseClicked(MouseEvent e), and draw(), updateScreen(), and paintComponent().

ExistsAI: boolean

If true, then the board controller initializes an AI enabled game.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by loadGameMouseClicked(MouseEvent e).

AI_COLOR: int

Denotes the integer representation of the AI controlled piece's colour.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by loadGameMouseClicked(MouseEvent e).

ACCESS PROGRAMS

MenuView(int N)

Constructs a new menu view object to visually represent the model on the screen.

In pseudocode:

Instantiate play game and debug buttons
Instantiate mouseClicked method
Instantiate Box
If ExistsAI:
Instantiate an AI

getState():int

Return the state of the application

playGameMouseClicked(MouseEvent e): void

If the mouse was clicked on the play game button

Set BoardController to visible

debugMouseClicked(MouseEvent e): void

If the mouse was clicked on the debug button

Set DebugController to visible

draw(Graphics g): void

This method formats all of the required components to the menu.

Set font: $FontSize = Width \times Default\_Font\_Size / Default\_Screen\_Width$

updateScreen(): void

Updates the screen

paintComponent(Graphics g): void

This method paints all of the formatted components to the menu.

playGameAIMouseClicked(MouseEvent e)

When the mouse is clicked on the button labeled “Play Game with Computer”, launch 1 player mode.

loadGameMouseClicked(MouseEvent e)

Loads the game from the save file found in ./saveGame.txt

3.2.6 CLASS: PLAYER

This class models a player. Each player is determined by two integers. An integer that represents their color , and the number of pieces they have left to place.

IMPLEMENTATION

USES

None

TYPE

COLOR: int

The value that corresponds to the colour.

This variable is final.

numberOfUnplayedPieces: int

The number of pieces that have not been played on the board.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, getNumberOfUnplayedPieces(), and placePiece().

ACCESS PROGRAMS

Player(int color, int numberOfUnplayedPieces)

Constructor method that takes in two parameters, color and number of unplayed pieces.

getColor():int

Returns the color of the player.

getNumberOfUnplayedPieces():int

Returns the number of pieces the player has yet to place.

placePiece(): void

When the player places a piece decrement numberOfUnplayedPieces by 1

3.2.7 CLASS: POINT

Defines a mathematical representation of a circle using its x-coordinate and its y-coordinate.

IMPLEMENTATION

USES

None

VARIABLES

x,y: double

X and Y coordinates of the point object.

This variable is kept private to reduce coupling between modules, and implement information hiding. X is maintained by the constructor, getX(), and getIntX(). Y is maintained by the constructor, getY(), and getIntY()

ACCESS PROGRAMS

Point(double x, double y)

Point constructor using two parameters

getX(): double

Return X coordinate.

getY(): double

Return Y coordinate.

getIntX(): int

Return integer approximation of the X coordinate.

getIntY(): int

Return integer approximation of the Y coordinate.

getDistance(Point that): double

Return the distance between two point objects, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

3.2.8 CLASS: RECTANGLE

Defines a mathematical representation of a rectangle defined by the top left, top right, and bottom left corners.

IMPLEMENTATION

USES

Point

VARIABLES

topLeft, topRight, bottomRight: Point

Three points will be used to construct each rectangle.

This variable is kept private to reduce coupling between modules, and implement information hiding. topLeft is maintained by the constructor, getTopLeft(), and getIntWidth(). topRight is maintained by the constructor, getTopRight(), and getIntWidth(). bottomRight is maintained by the constructor, getBottomRight(), and getIntHeight();.

ACCESS PROGRAMS

getTopLeft(): Point

Return the top left point of the rectangle. Used for assistance in geometric methods.

getTopRight(): Point

Return the top right point of the rectangle. Used for assistance in geometric methods.

getBottomRight(): Point

Return the bottom right point of the rectangle. Used for assistance in geometric methods.

Rectangle(Point topLeft, Point topRight, Point bottomRight)

Constructor method with 3 parameters.

getIntWidth(): int

Returns the width of the rectangle. This will be used for assistance in properly scaling based on window size.

getIntHeight():int

Return the height of the rectangle, $w = |y_{top\ right} - y_{bottom\ right}|$. This will be used for assistance in properly scaling based on window size.

getTopLeftIntX(): int

Return the X coordinate of the top left corner. Parameter to draw the rectangle.

getTopLeftIntY(): int

Return the Y coordinate of the top left corner. Parameter to draw the rectangle.

getBottomLeft(): Point

Return the bottom left point, defined by $(x_{topleft}, y_{bottomright})$

3.2.9 CLASS: SCREEN

Declares a function that will be used in classes that extend from it. In this assignment, that would be the classes menuView, and boardView. It is a template for the views.

IMPLEMENTATION

USES

JPanel

VARIABLES

None

ACCESS PROGRAMS

updateScreen(): void

It updates the screen.

3.2.10 CLASS: BOARD

Creates the model used by other classes in the program to construct the board. The class is essentially used to allow access to specific properties of the Six Men's Morris board. Properties such as state, and the number of pieces.

IMPLEMENTATION

USES

None

VARIABLES

N: int

Number of squares needed for the board.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructors and getN().

NUM_PIECES_PER_LAYER: int

Amount of pieces per layer.

This variable is final.

pieces: int[]

Amount of positions to place pieces. This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, setPieces, setPieceState, getBoardState, getPieceState, millExists, onlyMillsLeft, and checkWinner.

private PiecesHistory: int[][]

This variable is used to record the history of the pieces moved.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by Board.setPieceState.

private counter: int

This variable is used to keeps track of the number of moves over a sequence of 8 moves, as one repetition is 4 moves by both players.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by Board.setPieceState.

private repeats: int

Record number of repeated moves.

This variable is kept private to reduce coupling between modules, and implement

information hiding. It is maintained by Board.setPieceState.

ACCESS PROGRAMS

Board(int N)

Determine the number of pieces

*pieces = N*NUM_PIECES_PER_LAYER*

Suppose [λ] is the empty array of nodes where each node is possibly a pointer to a coloured piece. Suppose also that all possible n_k nodes can be generated upon request. The program initializes this array of nodes through the following specified behaviour.

{P(N) : P(N) = boolean(N > 5 ^ N % 3 == 0), [λ], n_k }

*Procedure **Board(N)***

{[n_0, n_1, n_2, … , $n_{(8N/3)}$]}

Board(int N, int[] pieces)

Construct a custom board depending on pieces

setPieces(int[] pieces): void

Allow for access to number of squares, and piece array for custom functions.

getN(): int

Return the number of squares of the board

setPieceState(int number, int state): void

Set state, not started, play mode or debug mode.

getBoardState: int[]

Return the current state of the board.

getPieceState(int number): int

Return the current state of the piece (black, red or blue).

Number is an index that will help determine the piece state.

millExists(int i): int[]

Return the positions of the pieces that form a mill including the piece on index i.

Returns [-1,-1,-1] if no mill is found.

See next page for the tabular expression.

i > 7	x = 8			
i <= 7	x = 0			
i % 2 == 0	pieces[(i+1)%8 + x] == same_colour_as_i	i % 8 == 0	pieces[7+x] == same_colour_as_i	mill[0] = i mill[1] = i+1 mill[2] = 7 + x
			pieces[7+x]!= same_colour_as_i	mill[0] = -1 mill[1] = -1 mill[2] = -1
		i % 8 != 0	pieces[(i-1)%8 + x] == same_colour_as_i	mill[0] = i mill[1] = i+1 mill[2] = i-1
			pieces[(i-1)%8 + x] != same_colour_as_i	mill[0] = -1 mill[1] = -1 mill[2] = -1
	pieces[(i+1)%8 + x] != same_colour_as_i			mill[0] = -1 mill[1] = -1 mill[2] = -1
i % 2 != 0	pieces[(i+1)%8 +x] == same_colour_as_i	pieces[(i+2)%8+x] == same_colour_as_i		mill[0] = i mill[1] = i+1 mill[2] = i+2
		pieces[(i+2)%8+x] != same_colour_as_i	i % 8 == 1	pieces[7 + x] == same_colour_as_i mill[0] = i mill[1] = i-1 mill[2] = 7+x
				pieces[7 + x] != same_colour_as_i mill[0] = -1 mill[1] = -1 mill[2] = -1
			i % 8 != 1	pieces[(i-2)%8 + x] == same_colour_as_i mill[0] = i mill[1] = i-1 mill[2] = i - 2
				pieces[(i-2)%8 + x] != same_colour_as_i mill[0] = -1 mill[1] = -1 mill[2] = -1
	pieces[(i+1)%8 +x] != same_colour_as_i			mill[0] = -1 mill[1] = -1 mill[2] = -1

Table 3: millExists() tabular expression. Note that the first 2 rows denote possible states for the variable x.

onlyMillsLeft(int color): boolean

Loop through every pieces and check for existing mill on that piece. If the first element of the returned mill integer list is -1, then return false. Otherwise return true.

boardIsEqual(int[] board1, int[] board2): boolean

Check if the given two boards are of the same length. If not return false.

Then check if every node in the two boards are the same. Return false if there is any different nodes.

If the above two test passes, return true.

3.2.11 CLASS: BOARDCONTROLLER

This class creates a window with labels and buttons during the regular game- play process. User interaction with those buttons will call for updates in the view, which in turn changes the representative values in the model.

IMPLEMENTATION

USES

JFrame

BoardView

Player

JLabel

AI

VARIABLES

NUMBER_OF_PIECES = 6: int

Denotes the number of pieces at each player's disposal for the Morris Game.

This variable is final.

BLUE_STATE: int

Used in allocating player turns.

This variable is final.

RED_STATE: int

Used in allocating player turns.

This variable is final.

FONT_SIZE: int

The default size of font assigned to view components.

This variable is final.

DEFAULT_SCREEN_WIDTH: int

The default screen width assigned to view components.

This variable is final.

DEFAULT_SCREEN_HEIGHT: int

The default screen height assigned to view components.

This variable is final.

jFrame: JFrame

Holds view components.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, resizeText, MouseClickEventHandler.mousePressed(), and removePiece.

boardView: BoardView

Holds a visualization of the board state.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor and resizeText().

turn: int

Assigns player turns. If turn int equals zero then system allocates next move or place to blue, else if turn equals one then system allocates next move to red.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, updateTitleColour(), placePieceState() and MouseClickEventHandler.mousePressed().

blue, red: Player

These are information containers assigned to each Player.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, updateLables(), updateState(), and placePieceState(), .

state: int

Holds current state of the game.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, updateState(), updateTitleText(), and MouseClickEventHandler.mousePressed().

blueLabel, blueCount, redLabel, redCount: JLabel

Used to organize and place visual objects and present visual components.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, resizeText(), and updateLabels().

title: JLabel

Used to organize and place visual objects and present visual components.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, updateTitleColour(), and updateTitleText().

selectedColour: int

Denotes the current player’s turn.

This variable is kept private to reduce coupling between modules, and implement

information hiding. It is maintained by MouseClickEventHandler.mousePressed().

selectedPiece: int

Represents the user's current selected piece.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by MouseClickEventHandler.mousePressed().

removePiece: boolean

Denotes the current piece selected.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by placePieceState(), removePiece(), and MouseClickEventHandler.mousePressed().

saveGame: JButton

Used to confirm a save request by the user.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor.

maxNumberOfRepeats = 3: int

Used in checking the draw condition in which game can not meaningfully

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by updateState() and MouseClickEventHandler.mousePressed().

ExistsAI: boolean

This variable is set to true if computer AI is enabled.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is used by BoardController(), update(), updateAIButton(), MouseClickEventHandler.mousePressed(MouseEvent e), MouseClickEventHandler.saveGameMouseClicked(MouseEvent e), MouseClickEventHandler.makeAIMoveMouseClicked(MouseEvent e) and it is maintained by initAI() in BoardController.

AI:AI

Used in calculating and determining the next move on the computer's turn when the game is in vs. Computer mode.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by initAI(int AI_colour) and BoardController().

AI_TURN: int

The integer representation of the AI's turn.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by initAI(), updateAIButton(), MouseClickEventHandler.mousePressed(MouseEvent e), and MouseClickEventHandler.makeAIMoveMouseClicked(MouseEvent e).

AI_COLOUR: int

The integer representation of the colour of the AI controlled pieces.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by initAI(int AI_colour) and MouseClickEventHandler.saveGameMouseClicked(MouseEvent e).

PLAYER_COLOUR: int

The integer representation of the colour of the human player controlled pieces. This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by initAI(int AI_colour) and MouseClickEventHandler.saveGameMouseClicked(MouseEvent e).

makeAIMove: JButton

Used to tell the controller to initiate the AI's turn.
This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by BoardController(), initAI(int AI_colour), and updateAIButton().

ACCESS PROGRAMS

BoardController(int N)

Framework for the game, manipulates the model to know how to update the view. The following is an outline of the program control flow;

Instantiates Random Turn → Instantiates Models → Instantiates Views

BoardController(int N, int[] boardState)

Construct a board controller with on a specified boardState class. These constructors automatically adds all of the pieces for each Player.

for i in range length of boardState
if boardState[i]==blue
bluePieces increment by 1
else if baordState[i] == red
redPieces increment by 1

BoardController(int N, int[] boardState, int turn, int state)

Constructs the board controller with a specified BoardState, turn integer, and state integer. Updates tile text and tile colours on creation.

BoardController(int N, boolean ExistsAI)

Constructs the BoardController with AI enabled.

update(boolean updateState): void

Updates state, view labels, title colours, title text strings and view.

resizeText(): void

Resize the text based on the dimensions of the window. This allows for dynamic change, such that the user can play with any size of window.

Set font: FontSize = Width×Default Font Size/Default Screen Width.

updateLabels(): void

This method will update the labels of each player involved.

noPossibleMoves(int colour): boolean

This method scans the board for a given colour, and returns true if there are no possible moves for that colour. It returns false otherwise.

updateView(): void

Controller takes information from the view and calls methods from the java.awt library on it. Repaints if it has been changed. Uses the MouseClickEventHandler which implements the MouseListener component. This class contains all possible methods that may be used for the Six Men's Morris Game. If in the future another MouseListener was to be used methods, the change is easily accommodated.

updateTitleColour():void

Updates the upper string of text to instruct that provides the player useful information such as what phase of the game is being played; placing phase or moving phase.

updateState(): void

Updates the current state value on this turn.

updateTitleText(): void

Updates the colour and text of the helper display above the board.

placePieceState(int i): void

Sends a request to place piece on the board to the model; updates the view accordingly.

Suppose player blue has $[b_0,b_1,b_2,...,b_n]$ pieces on the board, player red has $[r_0,r_1,r_2,...,r_n]$ number of pieces on the board and M is the number of pieces allotted for both players in the placing phase. Suppose that c denotes the coordinate to which the piece is to be assigned and V(c) is the function that validates whether the piece can be placed at c. Then if placePiece(int i) is called on player blue's turn, the program exhibits the following specified behaviour:

$\{V(c) \wedge n+1 < M \wedge \neg(\exists(b_{n+1}))\}$
*Procedure **placePiece(c)***
$\{[b_0,b_1,b_2,...,b_n,b_{n+1}]\}$

And if placePiece(int i) is called on player red's turn, the program should exhibits the following specified behaviour:

$\{V(c) \wedge n+1 < M \wedge \neg(\exists(r_{n+1}))\}$
*Procedure **placePiece(c)***
$\{[r_0,r_1,r_2,...,r_n,r_{n+1}]\}$

removedPiece(int i): void

Removes the currently selected piece.

Suppose player blue has $[b_0,b_1,b_2,...,b_n]$ pieces on the board and player red has $[r_0,r_1,r_2,...,r_n]$ number of pieces on the board. Then if removePiece(int i) is called on player red's turn, the program exhibits the following specified behaviour:

$\{[b_0,b_1,b_2,...,b_n]\}$
*Procedure **removePiece(b_n)***
$\{[b_0,b_1,b_2,...,b_{(n-1)},b_{(n+1)}]\}$

And if removePiece(int i) is called on player blue's turn, the program should exhibits the following specified behaviour:

{[r_0,r_1,r_2,...,r_n]}
Procedure ***removePiece(r_n)***
{[$r_{0,1}$,r_2,...,$r_{(n-1)}$,$r_{(n+1)}$]}

saveGameMouseClicked(MouseEvent e): void

Updates the current state when based on whether the current piece is set or removed, the current mode the user is in (1 player or 2 players), and if the user is in 1 player mode, record the colour of the AI.

mousePressed(MouseEvent e): void

This method is part of the private MouseClickedEventHandler class and allows for alternate colour pieces to be placed on the board after each click. The following is pseudocode for the behaviour of the method under different circumstances.

```
for i in range of length circles if mouse clicks circles[i]
    if state==0 switch turn%2
        case 0:
            if pieceNotTaken at i on boardView
                if unplayed blue pieces > 0
                    set that spot at i as 2 (red)
                    place blue piece
                increment turn
        case 1:
            if pieceNotTaken at i on boardView
                if unplayed red pieces > 0
                    set that spot at i as 2 (blue)
                    place red piece
                    decrement turn
                    update labels and view
                if number of red and blue unplayed pieces == 0
                    state =1
                else
                    state==1
```

Player_State = Red	Circle[i] = Pressed	Circle[i].state = Black	Set Circle[i].state = Red
		Circle[i].state != Black	Do nothing
	Circle[i] != Pressed		Do nothing
Player_State = Blue	Circle[i] = Pressed	Circle[i].state = Black	Set Circle[i].state = Blue
		Circle[i].state != Black	Do nothing
	Circle[i] != Pressed		Do nothing

Table 3: mousePressed() tabular expression

The following tabular expression specifies when the mouse click on Board is processed according to the ExistsAI variable.

ExistsAI == true	turn == AI_TURN	Click Processed = False
	turn != AI_TURN	Click Processed = True
ExistsAI == false	turn == AI_TURN	Click Processed = True
	turn != AI_TURN	Click Processed = True

Table 4: mousePressed() AI behaviour tabular expression

initAI(int AI_color): void

Initializes the AI for this game and assigns the AI controlled pieces the colour AI_color. If no colour is given then a randomly determined colour is assigned to the AI pieces. Also changes the text in makeAIMove button to tell user what the colour of the AI's pieces is.

updateAI(): void

Updates the AI's boardView by giving the AI class the boardView belonging to this Controller.

updateAIButton(): void

Disables the makeAIMove button if it is the player's turn.

makeAIMoveMouseClicked(MouseEvent e): void

Processes the AI move event. Calls different APIs from the AI class for different game states and uses the returned values and positions to place a piece, move a piece or remove a piece.

3.2.12 CLASS: BOARDVIEW

Creates the information that the controller will access in order to communicate to the user. The controller will call the view and this is what will draw the graphics to the application window.

INTERFACE

USES

Screen

Board

Circle[]

VARIABLES

board: Board

Represent the the Board object from Model.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, setBoardState,

getBoardState, millExists, existsOnlyMills, checkWinner, and getRepeats.

N: int

Number of squares.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor and draw().

states: int[]

Array of integers that holds each state.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, pieceNotTaken, getBoardStates, setBoardStates, and drawCircle.

COLORS = { Color.BLACK, Color.BLUE, Color.RED }: Color[]

If a third colour was introduced, add it here.

This variable is final.

RECTANGLE_WIDTH_SCALING = 0.19: double

Rectangle width scaling.

This variable is final.

RECTANGLE_HEIGHT_SCALING = 0.19: double

Rectangle height scaling.

This variable is final.

CIRCLE_SCALING = 0.07: double

Circle scaling.

This variable is final.

HORIZONTAL_LINE_SCALING

= (CIRCLE_SCALING/RECTANGLE_WIDTH_SCALING)*0.9: double

Horizontal line scaling.

This variable is final.

VERTICAL_LINE_SCALING

=(CIRCLE_SCALING/RECTANGLE_HEIGHT_SCALING)*0.9: double

Vertical line scaling.

This variable is final.

circles: Circle[]

Array of circles , will be used multiple times.

This variable is kept private to reduce coupling between modules, and implement information hiding. It is maintained by the constructor, getCircles, and drawBoardCircles.

ACCESS PROGRAMS

BoardView(int N)

Construct a board from the board model, where N is the number of pieces (squares).

BoardView(int N, int[] boardState)

Construct the board from the model using a current state.

pieceNotTaken(int number): boolean

This method will allow us to make sure the user can only place one piece per node on the board. It will return a boolean value that determines if a piece is already places in location FALSE, a piece is already there.

getBoardStates(): int[]

Return the state of the board (player Red or player Blue).

setBoardState(int number, int state): void

Set the state of the board. Number is the specific index of the array of states. State will be the previous state of the board and will be updated

getBoardState(int number): int

Returns the current state of the board from accessing getPieceState. Number is the number used to index the array of states.

getCircles(): Circle[]

Return the array of all circles in the board

setState(int[] states): void

Set the states of the game. States is the array of states (black, red, blue).

draw(Graphics g): void

Draw the entire board.

drawRectangle(Graphics g, Rectangle rect): void

Draws a black rectangle (layer / square) that updates based on window size.

drawCircle(Graphcis g, Circle circle, int state): void

Draws a coloured circle based on current state of the board.switch states[state]

case 0:

set colour to black

case 1:

set colour to blue

case 2:

set colour to red

default:

set colour to green

drawLine(Graphics g, Point a, Point b): void

Draw a line from point a to b. This will be used to connect layers.

drawBoardCircles(Graphics g, Rectangle rect, int layer): void

Draw all circles needed for the board

Instantiate points → Draws Circles

drawMiddleLines(Graphics g, Rectangle rect): void

Connect the layers of the board together through the midpoints of inner layers.

*diameterWidth = rectangle width * (0.07/0.19)*0.9*

diameterHeight = rectangle height(0.07/0.19)0.9*

Instantiate points → Draw lines

drawBoard(Graphics g, int N): void

Draw the entire board based on predetermined scaling constants.

Instantiate Points

for i in range of length N

New Rectangle object

if i<(N-1)

drawMiddleLines

drawRectangle

drawBoardCircles

updateScreen(): void

Updates the screen.

paintComponent(Graphics g): void

Draw sections of the board only when they need to be.

millExists(int i): boolean

Returns whether a mill including the piece at index i exists by querying the Board model.

existsOnlyMills(int colour): boolean

Returns whether there only exists mills on the board with the specified colour by querying the Board model.

checkWinner(): int

Returns whether or not there is a winner in the game by querying the Board model: 0 if there is no winner, 1 if the winner is blue, 2 if the winner is red.

getRepeats():int

Returns the number of repetitions by querying the Board model.

3.2.13 CLASS: GAME

Launches the menu.

INTERFACE

TYPE

None

VARIABLES

None

ACCESS PROGRAMS

main(String[] args): void

Main method that calls the controller constructor. This makes the menu appear. Create a new menuController object and set to visible.

3.2.14 CLASS: AI

Controls the actions of the second player when in vs. computer mode.

INTERFACE

TYPE

None

VARIABLES

boardView: BoardView

Used in updating the game's view. When the AI class calculates actions for the the second player's turn, it provides methods to update the boardView. This variable references the already initialized boardView and makes changes accordingly.

PLAYER_COLOR: int

The integer representation of the colour of the pieces controlled by player.

AI_COLOR:int

The integer representation of the colour of the pieces controlled by this AI class.

ACCESS PROGRAMS

AI(BoardView board, int ai, int player)

Constructs the AI class.

updateBoardView(BoardView boardView): void

Updates the view to the most recent view.

nextPlace():int

Finds the next available position to place a piece. Tries to allocate a piece to a node in the middle points first, then allocates pieces instead to the the corner nodes.

nextRemove(): int

Finds the next available piece belonging to the player to remove if and only if a mill was detected on this AI turn. The AI will remove first a piece from a Player's mill. If no mill was found, then it will remove the Player's next available piece.

nextMove(): int[]

Returns the integer array denoting the next move. Finds a piece on the board that is available to move, then moves it appropriately. Loops through the entire array of available pieces, then moves the first available piece in a random direction.

The returned array is in an [A,B] format where;

A - denotes the piece to to move.

B - denotes the place to which this piece should be moved to.

4 Trace to Requirements

The table below lists the assignment's requirements, and the modules that fulfil those requirements.

Requirements	Modules	How is it achieved
Assignment 1		
Enable the user to set up a board to play the game.	Board, BoardController, BoardView	The user initiate the game through clicking the JLable from BoardView, which calls the BoardController to create a new board from Board.
The board includes two types of discs.	BoardController, BoardView	BoardController sets up two different player with their own discs. BoardView sets up different colours for the two types of discs.
The discs are placed on either side of the board.	BoardController	BoardController has different JLables to display the number of discs on each side.
There are no discs at the start of the game.	BoardController, Board, BoardView	BoardController creates a Board object at the start of the game. Board is initialized with an empty array of discs. BoardView displays nodes with no disc as black.
The order of play is determined randomly.	BoardController	In the constructor, BoardController generates a random integer from 1 to 2 to determine whose turn it is.
The user should be able to start a new game, or enter discs to represent the current state of a game.	MenuController, MenuView	User can interact with JLables on MenuView, which will call MenuController for corresponding actions.
The user should be able to enter discs to represent the current state of a game by selecting a colour and clicking on the position of the disc.	DebugController, Board, BoardView	The user mouse click is received by DebugController, which will modify the Board, and update BoardView correspondingly.
When all the discs the user wants to play have been played, the system should analyze whether the current state is possible.	DebugController	In DebugController, boardIsLegal will be called to determine whether the current board state is legal.
Errors should be displayed to the user.	ErrorDialog	ErrorDialog extends JDialog, and the constructor will take a JFrame to be displayed in, a string for title, and string

		for message. Then it will be displayed onto the JFrame by this.setVisible(true).
Assignment 2		
Enable user to place pieces in turns	BoardController, BoardView, Board	In BoardController, placePieceState is used to determine which player's turn it is. And mousePressed will allow the corresponding user to place a disc.
All move has to be legal moves	DebugController, BoardController, BoardView, Board	Upon a mouse click event, the view and model will only be updated when boardIsLegal return true.
Determine which player won	Board, BoardView, BoardController	In Board.checkWinner, it will count the number of discs from each player, and return the player who has more than 2 discs.
Result of the game is displayed at all times	BoardController	In BoardController, a JLable is used to display state, and another variable, state, is used to represent the current state.
Order of the play is determined randomly.	BoardController	In the constructor, BoardController generates a random integer from 1 to 2 to determine whose turn it is.
User is able to choose to start a new game, store an existing unfinished game, and restart a stored game.	MenuView, BoardController	In MenuView, different JButton is created for new game, and continue from an existing one. Then the BoardController will be called upon mouse click of the above JButtons. Save game is handled by BoardController. Upon mouse click of the JButton for save game, BoardController will create "saveGame.txt" to store the board state.

Table 5: Traceability

5 Uses Relationship

The figure below shows the dependencies between the different modules. Arrows point from the user to the dependency.

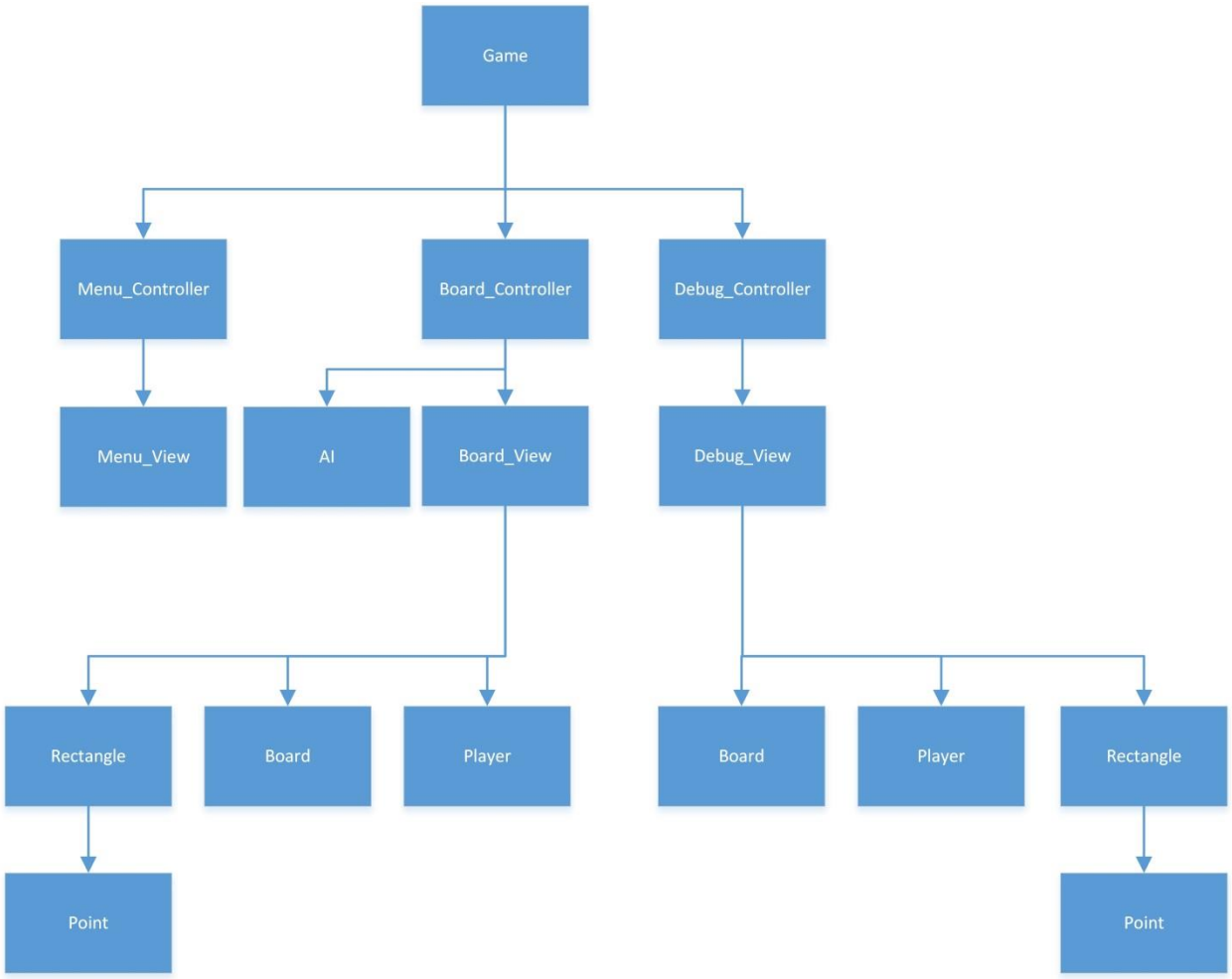

Figure 2: The Uses Relationship

From the diagram above, it is clear that the gameuses three different controllers: the menu controller, the board controller, and the debug controller. This relationship demonstrates that the game is made up of three different states, and it provides a point of entry to the program for a programmer implementing Six Men's Morris. The controllers each use a separate view, indicating that the controllers manipulate the view. This is also an important relationship, as it explicitly denotes that the controller controls the view and not vice versa. Furthermore, the board controller uses the AI in order to allow for the user to play against the computer. Finally, each view uses models. This relationship is crucial to the success of the program as it shows that the models reflect the views. This relationship is also crucial as the controller depends on the models to be reflective of the view in its state logic.

6 The AI Algorithm

6.1 Overview

The AI is implemented in a simple yet competitive manner. The implementation of the AI was divided into 2 segments: placing pieces, and moving pieces. In the first stage of the game, players place pieces onto the board. During this stage, the AI will aggressively place pieces by occupying the center positions. This will ensure that the AI has the highest likelihood of forming mills during this stage, as well as ensuring the most mobility during the latter stage. If all center positions are occupied, then the AI will occupy corner positions in order to form mills. If the AI achieved a mill during the placing stage, it will remove the first opponent piece it finds on the board.

During the second stage, each player sequentially moves their pieces. The AI will select the first piece it finds on the board of its colour, and determine if there are any legal moves for that piece. If there are two or three possible moves, the AI will determine the move randomly. If there are no possible moves for that piece, then the AI will examine to the next piece, repeating the above algorithm. If the AI were to form a mill at this stage, then the AI will remove the first opponent piece it finds on the board.

6.2 The Board Model

The notion of removing the first of the opponent's pieces, and selecting the first piece the AI finds is ill-defined without first introducing the Board model. In this implementation of Six Men's Morris, the board is modeled as a one dimensional array of integers. Each address in the array represents a location on the board, and each integer represents the colour of the piece that is on the board. A graphical representation of the board is given below:

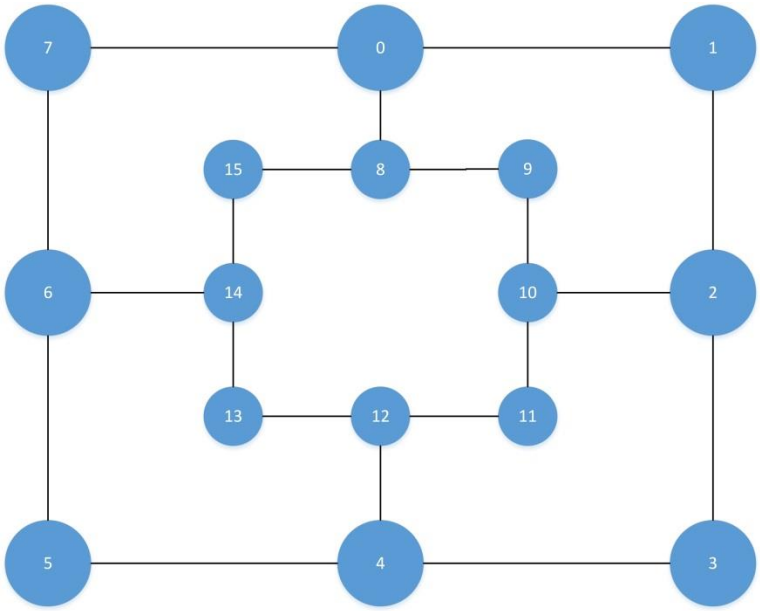

Figure 3: The Board

Note that the numbers in the figure above correspond to their array access location.

Such implementation of the board was used in order to optimize both space and time complexity with respect to the 3 dominant operations required of any Six Men's Morris board: scanning the board, accessing (or changing) the state of the board, and finding adjacent positions on the board. Two intuitive implementations of the Six Men's Morris board include using a 2 dimensional array, or using a bidirectional graph.

In the 2 dimensional array board model, a 5x5 integer array is created (for a 2 layer Six Men's Morris game), and certain locations in the array map to pieces on the board. The state of each piece is encoded into an integer in the array. In the worst case, finding a piece on the board could take quadratic, $O(N^2)$, time. This is because for every column in the array would have to be examined in every row of the array. Accessing the state of the board would take constant, O(1), time, as arrays support random access. A typical implementation of the 2 dimensional array board model would require quadratic, $O(N^2)$, time in order to find adjacent positions on the board, as the entire board would have to be scanned in order to find the adjacent pieces. However, there may exist a mathematical function that will allow connections to be determined in linear, O(N), time, on the basis that such a function was discovered for the one dimensional array implementation of the board model. Finally, storing this board would be of exponential, $O(2^N)$, complexity, as adding one additional layer to the board would require doubling the dimensions of the array. Additionally, not all of the allocated memory in the array is used: as the number of layers of the board increases, the amount of unused, allocated memory exponentially increases.

In the bidirectional graph board model, 16 nodes are created (for a 2 layer Six Men's Morris game), and the state of each piece as well as the connections between each piece is stored in the nodes. Scanning the board would be of linear, O(N), time, as either depth first search or breath first search can be used. Accessing a piece, however, would also be linear, O(N), time in the worst case, as the program would have to traverse the entire graph to get to the worst-case node. Finding adjacent positions would take constant, O(1), time, as the connection between nodes is stored in an array. Storing this implementation of the board would be of linear, O(N), complexity, as each node would have to be stored. Note that there is no unused allocated space with the bidirectional graph implementation of the board model.

The two intuitive implementations of the board model shows that there exists a way to implement a board model such that the access time would be constant, O(1), and there is a way to implement the board model such that scanning and storing the board would be of linear complexity, O(N). The one dimensional array implementation of the board model combines the constant access time provided by the 2 dimensional array implementation of the board model with the linear scanning and storage complexity provided by the bidirectional graph implementation of the board model in order to minimize time and space complexity. Scanning the board requires, in the worst case, a loop to iterate every element of the array, which takes linear, O(N), time. Accessing individual elements in the array take constant, O(1), time as arrays support random access. Finding adjacent positions on the board also takes linear time, as a mathematical function involving the modulo operation was discovered to produce a Boolean value that encodes whether a given position is adjacent to the selected position. Such function works by taking the modulo of the given array location and 8 (call this x), and the modulo of the selected position and 8 (call this y). If x is within 1 of y, or if x is within 7 of y and y is equal to 0, then the 2 positions are adjacent. This operation can be extended to find all adjacent positions

given a single position, and the number of layers of the board. Of course, the tradeoff for efficiency in the implementation is intuitive clarity in the logic, and the one dimensional array lacks the intuitive clarity that the other two implementations above have. Nevertheless, the lack of intuitive clarity can be compensated for with extensive documentation. Furthermore, by using such function, the need to store the connections between nodes is eliminated, saving space in the program. Finally, storing the array also takes linear, O(N), complexity, as adding 8 additional pieces (1 layer) would require adding 8 additional memory locations. Therefore, the one dimensional implementation of the board model was used in order to optimize both time and space efficiency.

The following table summarizes the time and space complexity of the different board implementations discussed above.

Implementation	Scanning	Accessing	Finding Connections	Space Complexity
Two Dimensional Array	$O(N^2)$	O(1)	$O(N^2)$	$O(2^N)$
Bidirectional Graph	O(N)	O(N)	O(1)	O(N)
One Dimensional Array	O(N)	O(1)	O(1)	O(N)

Table 6: Comparison of Time and Space Complexities

6.3 AI Interactions with the Board Model

With an established introduction to the board model, it is now appropriate to elaborate on certain operations of the AI described in *section 6.1 Overview*: how the AI mills, and how the AI selects pieces to move.

In order for the AI to mill, it starts a scan of the entire board. If the AI finds an opponent's piece, then it determines whether only mills exist for the opponent. If so, then then the piece the AI was examining is removed. Otherwise, the AI will continue to scan the board until another one of the opponent's pieces is found, and the process is repeated.

A similar method is used in order for the AI to select pieces to move. It first scans the board for its own pieces. When a piece is found, it uses the mathematical function described above to determine whether there is a legal move. If so, it takes the first available legal move. Otherwise, it continues on with its scan, repeating the process until a legal move is found.

7 Anticipated Changes & Discussion

In the design of Assignment 1, we anticipated the following changes:

- The BoardController can move between different states (i.e. setup, play, results)
- The player can play against the computer
- The application must efficiently store and search for a piece's next path
- There game can be expanded to N Men's Morris, where N is greater than 6
- Additional components can be added onto the BoardView and the DebugView
- Users can make an infinite number of moves
- The platform which to run the game will change over time
- The resolution of computer screens will change over time

The following subsections will examine each of the items above, and it will discuss how design decisions were made in Assignment 1 in order to accommodate for these changes.

7.1 There Can Be More States than the Defined States

The MenuController class allows for additional classes to be added to the application. It serves as a link to other states, so that an unforeseen state, such as the option to play another game besides N-Men's Morris can be implemented as another module within the same application.

7.2 The Player Can Play Against the Computer

In this application, the Player is modelled as an abstract data type. This means that a computer can be programmed to call commands that a human user could make using the mouse. Instead of solely relying on mouse input to place pieces on the board, mouse events trigger methods in the Player object, which in turn places a piece on the board.

7.3 The Application Must Efficiently Store and Search for a Piece's Next Path

The board in this application is implemented as a 1 dimensional array. This means that every element in the array will contain a value, so that no additional space is required. Also, the board is represented such that the adjacent nodes are beside each other in the array, or eight spots in front or behind. This makes checking the state of the board efficient, as a series of modular functions can be used in order to check the relevant pieces. The use of recursion can be avoided, as the traditional graph representation unused, hence saving both time and space.

7.4 The Game Can Be Expanded to N Men's Morris

The modules are implemented such that the number of layers (of rectangles) the board contains,

and the number of pieces each player has is contained in variables such as N and NUMBER_OF_PIECES. This makes modification of such parameters simple, as the rest of the code will remain constant.

7.5 Additional Components Can Be Added to the Views

Different components of the screen are encapsulated into JPanels which are then assembled in the controller. This allows for new components to be created as JPanels which can then replace existing components in the screen.

7.6 The Users Can Make an Infinite Number of Moves

The turn based system is implemented such that one player increments the turn counter while the other player decrements the turn counter. This makes the turn counter switch between 0 and 1. This allows for an infinite number of moves to be played while using as little space as possible. That is, the program will not crash if 2 computers decided to play against each other, and they use more than 232 moves.

7.7 The Platform Will Change Over Time

Java was the language of choice because it allowed for cross-platform integration of the application. Additionally, only the standard Java libraries were used in order to allow users to run the application with the minimal number of additional installations. This saves usage space, and it further prevents compatibility and licensing issues. The user of standard Java library is also, in our opinion, the best guarantee that the libraries used will be supported, as long as Java is supported.

7.8 The Resolution of Computer Screens Will Change Over Time

The program has been designed to fit screens of all shapes and sizes. The size of the components on the screen is based on the the screen’s width and height, and the user can resize the screen so that it fits comfortable on their monitor. The screen is rendered at a resolution of 500 x 500, which is small for 2016 standards, so that it can accommodate platforms with smaller screens, but it can be scaled indefinitely large for larger screens.

8 Test Plan/Design

8.1 Testing for Assignment 1

8.1.1 Requirement 1

Test #: 1
Units/Modules/Requirements: Enable the user to set up a board to play the game
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController and choose Start game.	Window with a board with 2 rectangles, within the other, with circles on the corners and at the midpoint of the lines. Lines connecting the middle circles of the big rectangle to the middle circles of the middle rectangle.

Table 7: Testing Requirement 1.1

Conclusion: This is the proper set-up for 6 Men's Morris.

8.1.2 Requirement 2

Test #: 2
Units/Modules/Requirements: The board includes 2 types of discs
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose start game, place discs on black circles.	Discs placed alternate between red and blue, the first colour determined randomly.

Table 8: Testing Requirement 1.2

Conclusion: There are red and blue discs

8.1.3 Requirement 3

Test #: 3

Units/Modules/Requirements: The discs are placed on either side of the board
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose start game, place discs on black circles.	On the left there is the amount of Blue discs remaining, on the right the amount of Red (beginning with 6 discs each). When a disc is place the number decreases depending on the colour of the disc placed.

Table 9: Testing Requirement 1.3

Conclusion: The amount of discs for each player is placed on the sides of the board. The amount of discs decreases as they are placed.

8.1.4 Requirement 4

Test #: 4
Units/Modules/Requirements: There are no discs at the start of the game
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose start game.	All the circles on the board are black, not red or blue.

Table 10: Testing Requirement 1.4

Conclusion: Black circles mean there are no discs placed on them. There are no discs at the start of the game.

8.1.5 Requirement 5

Test #: 5
Units/Modules/Requirements: The order of play is determined randomly
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose Start Game and place a piece anywhere.	Blue piece placed

Run MenuController, choose Start Game and place a piece anywhere.	Red piece placed
Run MenuController, choose Start Game and place a piece anywhere.	Blue piece placed
Run MenuController, choose Start Game and place a piece anywhere.	Blue piece placed

Table 11: Testing Requirement 1.5

Conclusion: The starting colour is not consistent; therefore, the starting colour is randomly decided every time.

8.1.6 Requirement 6

Test #: 6
Units/Modules/Requirements: The user should be able to start a new game, or enter discs to represent the current state of the game
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, click Start Game.	Menu with option of Start game or Debug. When Start game button clicked goes to game mode.
Run MenuController, click Debug.	Menu with option of Start game or Debug. When Debug chosen it gives the user the option of what colour to place and user is able to place pieces.

Table 12: Testing Requirement 1.6

Conclusion: Menu directs the user to either game mode or to place pieces and then start the game.

8.1.7 Requirement 7

Test #: 7
Units/Modules/Requirements: The user should be able to enter discs to represent the current state of a game by selecting a colour and clicking on the position of the disc
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController and choose Debug, choose colours and click circles.	Circles clicked change to the colour of the colour chosen from the menu on the left.

Table 13: Testing Requirement 1.7

Conclusion: The circles clicked change to the colour chosen and the user is able to enter the discs to represent a state of the game.

8.1.8 Requirement 8

Test #: 8
Units/Modules/Requirements: When all discs the user wants to play have been played, the system should analyse whether the current state is possible
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose Debug, place 4 blue pieces and 4 red pieces. Play game.	Game mode begins.
Run MenuController, choose Debug, place 1 blue pieces and 3 red pieces. Play game.	The user receives an error message.
Run MenuController, choose Debug, place 3 blue pieces and 1 red pieces. Play game.	The user receives an error message.
Run MenuController, choose Debug, place 10 blue pieces and 3 red pieces. Play game.	The user receives an error message.
Run MenuController, choose Debug, place 3 blue pieces and 10 red pieces. Play game.	The user receives an error message.

Table 14: Testing Requirement 1.8

Conclusion: When a legal amount of discs are placed, the player is able to play the game. When an illegal amount of discs is placed, the user receives an error message.

8.1.9 Requirement 9

Test #: 9
Units/Modules/Requirements: Errors should be displayed to the user
Test Type: Black Box (Functional Testing)

Input	Result
Run MenuController, choose Debug, place 10	Error window appears, there are too many

blue pieces and 6 red pieces. Play game.	pieces.
Press OK on error message, replace all pieces with black. Play game	Error window appears, both players have fewer than 3 pieces.

Table 15: Testing Requirement 1.9

Conclusion: When a user tries to make an impossible state the application tells them that it is illegal and why. The user is able to go back and change the discs to create a legal state.

8.2 Testing for Assignment 2

8.2.1 Requirement 1

Test #: 1
Units/Modules/Requirements: The player is able to make moves in turns
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Play a blue piece, and then a red piece	Blue piece played. Red piece played.	Blue piece played. Red piece played.	Pass
Play a red piece and then a blue piece	Red piece played. Blue piece played.	Red piece played. Blue piece played.	Pass
Play a blue piece to form a mill, remove a red piece, and then play a red piece	Blue piece played. Mill formed, red piece removed. Red piece played.	Blue piece played. Mill formed, red piece removed. Red piece played.	Pass
Play a red piece to form a mill, remove a blue piece, and then play a blue piece	Red piece played. Mill formed, blue piece removed. Blue piece played.	Red piece played. Mill formed, blue piece removed. Blue piece played.	Pass
Place a blue piece on the board. Then, place a red piece on the board.	Blue piece placed on board. Red piece placed on board.	Blue piece placed on board. Red piece placed on board.	Pass
Place a red piece on the board. Then place a blue piece on the board.	Red piece played on board. Blue piece played on board.	Red piece played on board. Blue piece played on board.	Pass

Table 16: Testing Requirement 2.1

8.2.2 Requirement 2

Test #: 2
Units/Modules/Requirements: The moves have to be legal moves
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Move top left most piece up (assume adjacent spots are empty).	ErrorDialog	ErrorDialog	Pass (This counts as a pass because the program behaved as expected).
Move top left most piece down (assume adjacent spots are empty).	Piece moves down	Piece moves down	Pass
Move top left most piece left (assume adjacent spots are empty).	ErrorDialog	ErrorDialog	Pass
Move top left most piece right (assume adjacent spots are empty).	Piece moves right	Piece moves right	Pass
Move bottom center piece up (assume adjacent spots are empty).	Piece moves up	Piece moves up	Pass
Move bottom center piece down (assume adjacent spots are empty).	ErrorDialog	Error dialog	Pass
Move bottom center piece left (assume adjacent spots are empty).	Piece moves left	Piece moves left	Pass

Move bottom center piece right (assume adjacent spots are empty).	Piece moves right	Piece moves right	Pass
Move piece in top center of inner layer up to an already filled spot.	ErrorDialog	ErrorDialog	Pass
Move piece in center right of inner to a non-adjacent empty square.	ErrorDialog	ErrorDialog	Pass
Move square in center right of outer layer to a non-adjacent filled square.	ErrorDialog	ErrorDialog	Pass

Table 17: Testing Requirement 2.2

8.2.3 Requirement 3

Test #: 3
Units/Modules/Requirements: The application has to recognize when the game has been won.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Win game with red player.	Game transitions to “Red Won”	Game transitions to “Red Won”	Pass
Win game with blue player.	Game transitions to “Blue Won”	Game transitions to “Blue Won”	Pass

Table 18: Testing Requirement 2.3

8.2.4 Requirement 4

Test #: 4

Units/Modules/Requirements: The game has to recognize when the game cannot be won.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Draw game on a red move.	Game transitions to “Game Drawn”	Game transitions to “Game Drawn”	Pass
Draw game on a blue move.	Game transitions to “Game Drawn”	Game transitions to “Game Drawn”	Pass
Trap blue so that blue has no legal moves.	Game transitions to “Red Win”	Game transitions to “Red Win”	Pass
Trap red so that red has no legal moves.	Game transitions to “Blue Win”	Game transitions to “Blue Win”	Pass

Table 19: Testing Requirement 2.4

8.2.4 Requirement 5

Test #: 5
Units/Modules/Requirements: The result of the game must be displayed at all times.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Start the game	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass
Enter place pieces stage	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass
Enter play game state	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass
Enter blue won state	JLabel displays “Blue Won”	JLabel displays “Blue Won”	Pass
Enter red won state	JLabel displays “Red Won”	JLabel displays “Red Won”	Pass

Enter game drawn state	JLabel displays “Game Drawn”	JLabel displays “Game Drawn”	Pass
Enter game using the Load Game button, from place piece stage.	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass
Enter game using the Load Game button, from play game stage.	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass
Enter game using the Load Game button, from red won stage.	JLabel displays “Red Won”	JLabel displays “Red Won”	Pass
Enter game using the Load Game button, from blue won stage.	JLabel displays “Blue Won”	JLabel displays “Blue Won”	Pass
Enter game using the Load Game button, from game drawn stage.	JLabel displays “Game Drawn”	JLabel displays “Game Drawn”	Pass

Table 20: Testing Requirement 2.5

8.2.6 Requirement 6

Test #: 6
Units/Modules/Requirements: Each move must be checked to show that it is legal.
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Move upper center piece on the outer layer to an empty adjacent square on the right.	Piece moves to the right	Piece moves to the right	Pass
Move center right piece on outer layer to an adjacent filled	ErrorDialog	ErrorDialog	Pass

square down.			
Move lower center piece on inner layer to a non-adjacent empty square up.	ErrorDialog	ErrorDialog	Pass
Move upper center piece in inner layer to a non-adjacent filled square left.	ErrorDialog	ErrorDialog	Pass
Red tries to move a blue piece.	ErrorDialog	ErrorDialog	Pass
Blue tries to move a red piece.	ErrorDialog	ErrorDialog	Pass
Red captures a blue piece not part of a mill.	Blue piece captured	Blue piece captured	Pass
Blue captures a red piece not part of a mill.	Red piece captured	Red piece captured	Pass
Red captures a blue piece that is part of a mill, when a blue piece that is not part of a mill exists.	ErrorDialog	ErrorDialog	Pass
Blue captures a red piece that is part of a mill when a red piece that is not part of a mill exists.	ErrorDialog	ErrorDialog	Pass
Red captures a blue piece that is part of a mill when only blue mills exist.	Blue piece captured	Blue piece captured	Pass
Blue captures a red piece that is part of a mill when only red mills exist.	Red piece captured	Red piece captured	Pass

Red tries to capture a red piece.	ErrorDialog	ErrorDialog	Pass
Blue tries to capture a blue piece.	ErrorDialog	ErrorDialog	Pass

Table 21: Testing Requirement 2.6

8.2.7 Requirement 7

Test #: 7
Units/Modules/Requirements: The order of play (blue first or red first) shall be determined randomly
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Start game until blue goes first	After n repetitions, blue goes first	After starting the game once, blue went first	Pass
Start game until red goes first.	After n repetitions, red goes first	After restarting the game 4 times, red went first.	Pass

Table 22: Testing Requirement 2.7

8.2.8 Requirement 8

Test #: 8
Units/Modules/Requirements: The user shall be able to start a new game, store an existing unfinished game, and restart a stored game
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Start a new game	Game starts	Game starts	Pass
Save a game in the place pieces state, on red's turn	Game saved	Game saved	Pass

Save a game in the place pieces state, on blue's turn	Game saved	Game saved	Pass
Load a game in the place pieces state, on red's turn	Game loads on red's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Load a game in the place pieces state, on blue's turn	Game loads on blue's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Save a game in the play state, on red's turn during a movement	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during a movement	Game saved	Game saved	Pass
Save a game in the play state, on red's turn during which red may capture a blue piece.	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during which blue may capture a red piece	Game saved	Game saved	Pass
Load a game in the play state, on red's turn during a movement	Game loaded on red's turn, and allowed red to move	Game loaded on red's turn, and allowed red to move	Pass
Load a game in the play state, on blue's turn during a movement	Game loaded on blue's turn, and allowed blue to move	Game loaded on blue's turn, and allowed blue to move	Pass
Load a game in the play state, on red's turn, during which	Game loaded on red's turn, and allowed red to mill.	Game loaded on red's turn, and allowed red to mill.	Pass

red can capture a blue piece.			
Load a game in the play state, on blue's turn, during which blue may capture a red piece.	Game loaded on blue's turn, and allowed blue to mill.	Game loaded on blue's turn, and allowed blue to mill.	Pass
Save a game on red win	Game saved	Game saved	Pass
Save a game on blue win	Game saved	Game saved	Pass
Save a game on game drawn	Game saved	Game saved	Pass
Load a game on red win	Game loaded on red win state	Game loaded on red win state	Pass
Load a game on blue win	Game loaded on blue win state	Game loaded on blue win state	Pass
Load a game on game drawn	Game loaded on game drawn state	Game loaded on game drawn state	Pass

Table 23: Testing Requirement 2.8

8.3 Testing for Assignment 3

8.3.1 Black Box Testing

8.3.1.1 Requirement 1

Test #: 1
Units/Modules/Requirements: The user should be able to choose between two modes of operation: 2 player Six Men's Morris, in which 2 people can play against each other; or 1 player against the computer
Test Type: Black Box (Logic Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Click on Play Game (state = 2_Player)	Start 2 player mode	Started 2 player mode	Pass
Click on "Play Game	Start 1 player mode	Started 1 player mode	Pass

with Computer" (state = 1_Player)			
Do nothing (state = menu)	No result	No result	Pass
Click on both buttons (state = 1_Player ^ 2_Player)	Impossible since the states are mutually exclusive.	Started 2 player mode	Fail. However, this is acceptable since the program switched to the first state pressed.

Table 24: Testing Requirement 1

8.3.1.2 Requirement 2

Test #: 2
Units/Modules/Requirements: In 2 player mode, the behavior is the same as in assignment 2.
Test Type: Black Box (Decision Table-Based Testing)

8.3.1.2.1 Requirement 1

Test #: 2.1
Units/Modules/Requirements: The player is able to make moves in turns
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Play a blue piece, and then a red piece	Blue piece played. Red piece played.	Blue piece played. Red piece played.	Pass
Play a red piece and then a blue piece	Red piece played. Blue piece played.	Red piece played. Blue piece played.	Pass
Play a blue piece to form a mill, remove a red piece, and then play a red piece	Blue piece played. Mill formed, red piece removed. Red piece played.	Blue piece played. Mill formed, red piece removed. Red piece played.	Pass
Play a red piece to form a mill, remove a	Red piece played. Mill formed, blue	Red piece played. Mill formed, blue	Pass

blue piece, and then play a blue piece	piece removed. Blue piece played.	piece removed. Blue piece played.	
Place a blue piece on the board. Then, place a red piece on the board.	Blue piece placed on board. Red piece placed on board.	Blue piece placed on board. Red piece placed on board.	Pass
Place a red piece on the board. Then place a blue piece on the board.	Red piece played on board. Blue piece played on board.	Red piece played on board. Blue piece played on board.	Pass

Table 25: Testing Assignment 2, Requirement 2.1

8.3.1.2.2 Requirement 2

Test #: 2.2
Units/Modules/Requirements: The moves have to be legal moves
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Move top left most piece up (assume adjacent spots are empty).	ErrorDialog	ErrorDialog	Pass (This counts as a pass because the program behaved as expected).
Move top left most piece down (assume adjacent spots are empty).	Piece moves down	Piece moves down	Pass
Move top left most piece left (assume adjacent spots are empty).	ErrorDialog	ErrorDialog	Pass
Move top left most piece right (assume adjacent spots are empty).	Piece moves right	Piece moves right	Pass
Move bottom center	Piece moves up	Piece moves up	Pass

piece up (assume adjacent spots are empty).			
Move bottom center piece down (assume adjacent spots are empty).	ErrorDialog	Error dialog	Pass
Move bottom center piece left (assume adjacent spots are empty).	Piece moves left	Piece moves left	Pass
Move bottom center piece right (assume adjacent spots are empty).	Piece moves right	Piece moves right	Pass
Move piece in top center of inner layer up to an already filled spot.	ErrorDialog	ErrorDialog	Pass
Move piece in center right of inner to a non-adjacent empty square.	ErrorDialog	ErrorDialog	Pass
Move square in center right of outer layer to a non-adjacent filled square.	ErrorDialog	ErrorDialog	Pass

Table 26: Testing Assignment 2, Requirement 2.2

8.3.1.2.3 Requirement 3

Test #: 2.3
Units/Modules/Requirements: The application has to recognize when the game has been won.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**

Win game with red player.	Game transitions to “Red Won”	Game transitions to “Red Won”	Pass
Win game with blue player.	Game transitions to “Blue Won”	Game transitions to “Blue Won”	Pass

Table 27: Testing Assignment 2, Requirement 2.3

8.3.1.2.4 Requirement 4

Test #: 2.4
Units/Modules/Requirements: The game has to recognize when the game cannot be won.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Draw game on a red move.	Game transitions to “Game Drawn”	Game transitions to “Game Drawn”	Pass
Draw game on a blue move.	Game transitions to “Game Drawn”	Game transitions to “Game Drawn”	Pass
Trap blue so that blue has no legal moves.	Game transitions to “Red Win”	Game transitions to “Red Win”	Pass
Trap red so that red has no legal moves.	Game transitions to “Blue Win”	Game transitions to “Blue Win”	Pass

Table 28: Testing Assignment 2, Requirement 2.4

8.3.1.2.5 Requirement 5

Test #: 2.5
Units/Modules/Requirements: The result of the game must be displayed at all times.
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Start the game	JLabel displays “Game in Progress”	JLabel displays “Game in Progress”	Pass

Enter place pieces stage	JLabel displays "Game in Progress"	JLabel displays "Game in Progress"	Pass
Enter play game state	JLabel displays "Game in Progress"	JLabel displays "Game in Progress"	Pass
Enter blue won state	JLabel displays "Blue Won"	JLabel displays "Blue Won"	Pass
Enter red won state	JLabel displays "Red Won"	JLabel displays "Red Won"	Pass
Enter game drawn state	JLabel displays "Game Drawn"	JLabel displays "Game Drawn"	Pass
Enter game using the Load Game button, from place piece stage.	JLabel displays "Game in Progress"	JLabel displays "Game in Progress"	Pass
Enter game using the Load Game button, from play game stage.	JLabel displays "Game in Progress"	JLabel displays "Game in Progress"	Pass
Enter game using the Load Game button, from red won stage.	JLabel displays "Red Won"	JLabel displays "Red Won"	Pass
Enter game using the Load Game button, from blue won stage.	JLabel displays "Blue Won"	JLabel displays "Blue Won"	Pass
Enter game using the Load Game button, from game drawn stage.	JLabel displays "Game Drawn"	JLabel displays "Game Drawn"	Pass

Table 29: Testing Assignment 2, Requirement 2.5

8.3.1.2.6 Requirement 6

Test #: 2.6
Units/Modules/Requirements: Each move must be checked to show that it is legal.
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Move upper center piece on the outer layer to an empty adjacent square on the right.	Piece moves to the right	Piece moves to the right	Pass
Move center right piece on outer layer to an adjacent filled square down.	ErrorDialog	ErrorDialog	Pass
Move lower center piece on inner layer to a non-adjacent empty square up.	ErrorDialog	ErrorDialog	Pass
Move upper center piece in inner layer to a non-adjacent filled square left.	ErrorDialog	ErrorDialog	Pass
Red tries to move a blue piece.	ErrorDialog	ErrorDialog	Pass
Blue tries to move a red piece.	ErrorDialog	ErrorDialog	Pass
Red captures a blue piece not part of a mill.	Blue piece captured	Blue piece captured	Pass
Blue captures a red piece not part of a mill.	Red piece captured	Red piece captured	Pass
Red captures a blue piece that is part of a mill, when a blue piece that is not part of a mill exists.	ErrorDialog	ErrorDialog	Pass
Blue captures a red piece that is part of a mill when a red piece that is not part of a mill exists.	ErrorDialog	ErrorDialog	Pass

Red captures a blue piece that is part of a mill when only blue mills exist.	Blue piece captured	Blue piece captured	Pass
Blue captures a red piece that is part of a mill when only red mills exist.	Red piece captured	Red piece captured	Pass
Red tries to capture a red piece.	ErrorDialog	ErrorDialog	Pass
Blue tries to capture a blue piece.	ErrorDialog	ErrorDialog	Pass

Table 30: Testing Assignment 2, Requirement 2.6

8.3.1.2.7 Requirement 7

Test #: 2.7
Units/Modules/Requirements: The order of play (blue first or red first) shall be determined randomly
Test Type: Black Box (Functional Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Start game until blue goes first	After n repetitions, blue goes first	After starting the game once, blue went first	Pass
Start game until red goes first.	After n repetitions, red goes first	After restarting the game 4 times, red went first.	Pass

Table 31: Testing Assignment 2, Requirement 2.7

8.3.1.2.8 Requirement 8

Test #: 2.8
Units/Modules/Requirements: The user shall be able to start a new game, store an existing unfinished game, and restart a stored game
Test Type: Black Box (Functional Testing)

Case (Input)	Expected	Actual	Pass/Fail
Start a new game	Game starts	Game starts	Pass
Save a game in the place pieces state, on red's turn	Game saved	Game saved	Pass
Save a game in the place pieces state, on blue's turn	Game saved	Game saved	Pass
Load a game in the place pieces state, on red's turn	Game loads on red's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Load a game in the place pieces state, on blue's turn	Game loads on blue's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Save a game in the play state, on red's turn during a movement	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during a movement	Game saved	Game saved	Pass
Save a game in the play state, on red's turn during which red may capture a blue piece.	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during which blue may capture a red piece	Game saved	Game saved	Pass
Load a game in the play state, on red's turn during a movement	Game loaded on red's turn, and allowed red to move	Game loaded on red's turn, and allowed red to move	Pass

Load a game in the play state, on blue's turn during a movement	Game loaded on blue's turn, and allowed blue to move	Game loaded on blue's turn, and allowed blue to move	Pass
Load a game in the play state, on red's turn, during which red can capture a blue piece.	Game loaded on red's turn, and allowed red to mill.	Game loaded on red's turn, and allowed red to mill.	Pass
Load a game in the play state, on blue's turn, during which blue may capture a red piece.	Game loaded on blue's turn, and allowed blue to mill.	Game loaded on blue's turn, and allowed blue to mill.	Pass
Save a game on red win	Game saved	Game saved	Pass
Save a game on blue win	Game saved	Game saved	Pass
Save a game on game drawn	Game saved	Game saved	Pass
Load a game on red win	Game loaded on red win state	Game loaded on red win state	Pass
Load a game on blue win	Game loaded on blue win state	Game loaded on blue win state	Pass
Load a game on game drawn	Game loaded on game drawn state	Game loaded on game drawn state	Pass

Table 32: Testing Assignment 2, Requirement 2.8

8.3.1.3 Requirement 3

Test #: 3
Units/Modules/Requirements: In playing against the computer, the same mechanism as 2 player mode should be used to decide who plays first, once it has been decided whether the computer plays blue or red.
Test Type: Black Box (Decision Table-Based Testing)

Case (Input)	Expected	Actual	Pass/Fail
Run game until the following state is triggered: computer = blue player = first	Player goes first. Computer is blue.	Player goes first. Computer is blue.	Pass
Run game until the following state is triggered: computer = blue player = second	Player goes second. Computer is blue.	Player goes second. Computer is blue.	Fail, red always goes first. -- Pass
Run game until the following state is triggered: computer = red player = first	Player goes first. Computer is red.	Player goes first. Computer is red.	Fail, red always goes first. -- Pass
Run game until the following state is triggered: computer = red player = second	Player goes second. Computer is red.	Player goes second. Computer is red.	Pass

Table 33: Testing Requirement 3.2

8.3.1.4 Requirement 4

Test #: 4
Units/Modules/Requirements: The user should be able to choose to start a new game, store an existing unfinished game, and re-start a stored game.
Test Type: Black Box (Decision Table-Based Testing)

Case (Input)	Expected	Actual	Pass/Fail
Start a new game	Game starts	Game starts	Pass
Save a game in the place pieces state, on red's turn	Game saved	Game saved	Pass
Save a game in the place pieces state, on	Game saved	Game saved	Pass

blue's turn			
Load a game in the place pieces state, on red's turn	Game loads on red's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Load a game in the place pieces state, on blue's turn	Game loads on blue's turn with appropriate number of pieces	Game loads on red's turn with appropriate number of pieces	Pass
Save a game in the play state, on red's turn during a movement	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during a movement	Game saved	Game saved	Pass
Save a game in the play state, on red's turn during which red may capture a blue piece.	Game saved	Game saved	Pass
Save a game in the play state, on blue's turn, during which blue may capture a red piece	Game saved	Game saved	Pass
Load a game in the play state, on red's turn during a movement	Game loaded on red's turn, and allowed red to move	Game loaded on red's turn, and allowed red to move	Pass
Load a game in the play state, on blue's turn during a movement	Game loaded on blue's turn, and allowed blue to move	Game loaded on blue's turn, and allowed blue to move	Pass
Load a game in the play state, on red's turn, during which red can capture a blue piece.	Game loaded on red's turn, and allowed red to mill.	Game loaded on red's turn, and allowed red to mill.	Pass

Load a game in the play state, on blue's turn, during which blue may capture a red piece.	Game loaded on blue's turn, and allowed blue to mill.	Game loaded on blue's turn, and allowed blue to mill.	Pass
Save a game on red win	Game saved	Game saved	Pass
Save a game on blue win	Game saved	Game saved	Pass
Save a game on game drawn	Game saved	Game saved	Pass
Load a game on red win	Game loaded on red win state	Game loaded on red win state	Pass
Load a game on blue win	Game loaded on blue win state	Game loaded on blue win state	Pass
Load a game on game drawn	Game loaded on game drawn state	Game loaded on game drawn state	Pass

Table 34: Testing Requirement 3.3

8.3.1.5 Requirement 5

Test #: 5
Units/Modules/Requirements: The computer's move has to be legal.
Test Type: Black Box (Decision Table-Based Testing)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Allow computer to place piece	Move is legal	Move is legal	Pass
Allow computer to move piece	Move is legal	Move is legal	Pass
Allow computer to form a mill	Move is legal	Move is legal	Pass
Allow computer to capture a piece	Move is legal	Move is legal	Pass

Table 35: Testing Requirement 3.4

8.3.2 White Box Testing

Note: Only selected requirements deemed appropriate for white box testing were tested.

8.3.2.1 Requirement 1

Test #: 6
Units/Modules/Requirements: The user should be able to choose between two modes of operation: 2 player Six Men's Morris, in which 2 people can play against each other; or 1 player against the computer
Test Type: White Box Testing (Path Coverage on MenuController.java)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Click on "Play Game"	Start 2 player mode	Started 2 player mode	Pass
Click on "Play Game with Computer"	Start 1 player mode	Started 1 player mode	Pass
Click on "Debug"	Start debug mode	Started debug mode	Pass
Click on "Load Game" with saved game in 2 player mode.	Start 2 player mode	Started 2 player mode	Pass
Click on "Load Game" with saved game in 1 player mode.	Start 1 player mode	Started 1 player mode	Pass

Table 36: Testing Requirement 1

8.3.2.2 Requirement 2

Test #: 7
Units/Modules/Requirements: In playing against the computer, the same mechanism as 2 player mode should be used to decide who plays first, once it has been decided whether the computer plays blue or red.
Test Type: White Box Testing (Path Coverage on AI.java, BoardController.java)

Case (Input)	Expected	Actual	Pass/Fail
AI_COLOUR = 1 turn = 1	Computer is blue, player goes first.	Computer is blue, player goes first.	Pass
AI_COLOUR = 2 turn = 0	Computer is red, player goes first.	Computer is red, player goes first.	Pass
AI_COLOUR = 1 turn = 0	Computer is blue, player goes second.	Computer is blue, player goes second.	Pass
AI_COLOUR = 2 turn = 1	Computer is red, player goes second.	Computer is red, player goes second.	Pass

Table 37: Testing Requirement 3.2

8.3.2.3 Requirement 3

Test #: 8
Units/Modules/Requirements: In playing against the computer, the user should be able to choose to start a new game, store an existing unfinished game, and restart a stored game.
Test Type: White Box Testing (Path Coverage on AI.java, BoardController.java)

Case (Input)	Expected	Actual	Pass/Fail
Saved game during player's turn, placing pieces (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during computer's turn, placing pieces (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during player's turn, placing pieces, in the middle of a mill (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during computer's turn, placing pieces, in the middle of a mill	Game saved	Game saved	Pass

(AI_COLOUR = 1).			
Saved game during player's turn to move (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during computer's turn to move (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during player's turn to mill (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during computer's turn to mill (AI_COLOUR = 1).	Game saved	Game saved	Pass
Saved game during player's turn, placing pieces (AI_COLOUR = 2).	Game saved	Game saved	Pass
Saved game during computer's turn, placing pieces (AI_COLOUR = 2).	Game saved	Game saved	Pass
Saved game during player's turn, placing pieces, in the middle of a mill (AI_COLOUR = 2).	Game saved	Game saved	Pass
Saved game during computer's turn, placing pieces, in the middle of a mill (AI_COLOUR = 2).	Game saved	Game saved	Pass
Saved game during player's turn to move (AI_COLOUR = 2).	Game saved	Game saved	Pass
Saved game when player won	Game saved	Game saved	Pass

(AI_COLOUR = 1)			
Saved game when computer won (AI_COLOUR = 1)	Game saved	Game saved	Pass
Saved game at draw (AI_COLOUR = 1)	Game saved	Game saved	Pass
Saved game when player won (AI_COLOUR = 2)	Game saved	Game saved	Pass
Saved game when computer won (AI_COLOUR = 2)	Game saved	Game saved	Pass
Saved game at draw (AI_COLOUR = 2)	Game saved	Game saved	Pass
Loaded game during player's turn, placing pieces (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during computer's turn, placing pieces (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during player's turn, placing pieces, in the middle of a mill (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during computer's turn, placing pieces, in the middle of a mill (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during player's turn to move (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during computer's turn to	Game loaded	Game loaded	Pass

move (AI_COLOUR = 1).			
Loaded game during player's turn to mill (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during computer's turn to mill (AI_COLOUR = 1).	Game loaded	Game loaded	Pass
Loaded game during player's turn, placing pieces (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during computer's turn, placing pieces (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during player's turn, placing pieces, in the middle of a mill (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during computer's turn, placing pieces, in the middle of a mill (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during player's turn to move (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during computer's turn to move (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during player's turn to mill (AI_COLOUR = 2).	Game loaded	Game loaded	Pass
Loaded game during	Game loaded	Game loaded	Pass

computer's turn to mill (AI_COLOUR = 2).			
Loaded game when player won (AI_COLOUR = 1)	Game loaded	Game loaded	Pass
Loaded game when computer won (AI_COLOUR = 1)	Game loaded	Game loaded	Pass
Loaded game at draw (AI_COLOUR = 1)	Game loaded	Game loaded	Pass
Loaded game when player won (AI_COLOUR = 2)	Game loaded	Game loaded	Pass
Loaded game when computer won (AI_COLOUR = 2)	Game loaded	Game loaded	Pass
Loaded game at draw (AI_COLOUR = 2)	Game loaded	Game loaded	Pass

Table 38: Testing Requirement 3.3

8.3.2.4 Requirement 4

Test #: 9
Units/Modules/Requirements: The computer's move has to be legal.
Test Type: White Box Testing (Path Coverage on AI.java, BoardController.java)

Case (Input)	**Expected**	**Actual**	**Pass/Fail**
Place piece where the computer would play next.	Computer's move is legal	Computer's move is legal	Pass
Place piece away from computer's next move.	Computer's move is legal	Computer's move is legal	Pass
Force computer to	Computer move is	Computer move is legal.	Pass

capture a piece, where a mill exists and a piece not in a mill exists.	legal.	Piece not in mill captured.	
Force computer to capture a piece, where only mills exist	Computer move is legal	Computer move is legal	Pass

Table 39: Testing Requirement 3.4

9 Conclusion

This report details the design decisions made and the implementation of assignment 3. Overall, we believe that this is a robust implementation of Six Men's Morris that exemplifies the principles of software engineering. The requirements of this application were first formally defined, and attempts at using tabular expressions and mathematical functions to model the problem were made. This lead to the inception of a board-checking algorithm which allows for the board to have a space complexity proportional to the number of pieces on the board while maintaining linear search time in the worst case. Furthermore, three separate controllers and views were developed from formally defining the requirements, which allowed each state of the application to remain independent of the other states. This allows our program to exhibit separation of concerns. Through the decomposition of our program, we were able to separate our program into three main areas, models, controllers, and views. This ultimately allowed us to create a program using the MVC architecture. The components of the MVC model in the implementation allowed for modularity and abstraction. ADTs were created which allowed for information hiding, low coupling between different states, and high cohesion. The creation of modules also allowed us to anticipate change. We were able to implement the assignment such that it could be easily modified to accommodate future changes. Both separation of concerns and modularity in our code allowed us to implement changes with minimal change to the overall interface of the code.

In particular, our thorough design of Six Men's Morris in Assignment 1 and 2 allowed for a seamless integration of the AI module to the game. The previous design decisions made it intuitive to design the logic behind what moves the AI should play, and what pieces the AI should take during a mill. Our linear mathematical representation allowed for linear time decision making, as the AI would only have to scan the board once before deciding on a move. Furthermore, the mathematical representation of our board allowed for easy verification and validation, as edge cases and typical cases could be easily identified. Additionally, the multitude of modules created beforehand allowed for the AI to be made and tested on an independent platform as a single dependent variable: it was easy to be confident that all other modules were correct to the extent of the specification, as they were tested in assignments 1 and 2.

However, our current design of the game does not take full advantage of the decompositions made in assignment 1 and 2. For instance, assignment 1 introduced a player model, which modeled the move of each player. This was done in anticipation of an AI module, so that it would be simple to place pieces, and to remove pieces. Nevertheless, the AI module does not use the player model. Yet, despite these drawbacks to the design of assignment 3, one can be confident that the application is correct to the level of the specification, due to extensive testing, and validation.

Therefore, since our implementation of assignment 3 embodies all of the principles of software engineering, we believe that our application is a robust implementation of the Six Men's Morris interactive game.

10 Appendix

10.1 Change log

- Added AI.java class documentation to MIS and MID. The addition can be found under section 3.1.14 CLASS: AI under MIS and section 3.2.14 CLASS: AI under MID.
- Added documentation for additional constructor for the BoardController. Changes found under section 3.1.11 CLASS: BOARDCONTROLLER and section 3.2.11 CLASS: BOARDCONTROLLER.
- Added documentation for the following private variables to the section 3.2.11 CLASS: BOARDCONTROLLER: boolean ExistsAI, AI AI, int AI_TURN, int AI_COLOR, int PLAYER_COLOR, JButton makeAIMove
- Changed the maxNumberOfRepeats to 12 in 3.2.11 CLASS: BOARDCONTROLLER.
- Added documentation for a new public method initAI() in section 3.1.11 CLASS: BOARDCONTROLLER and section 3.2.11 CLASS: BOARDCONTROLLER
- Changed documentation for update() method in section 3.2.11 CLASS: BOARDCONTROLLER
- Added documentation for a new private method updateAIButton() in section 3.2.11 CLASS: BOARDCONTROLLER
- Added documentation for a new private method removePieceButton(int i) in section 3.2.11 CLASS: BOARDCONTROLLER
- Added pre/post conditions for 3.2.11 CLASS: BOARDCONTROLLER for placePieceState() and removePiece()
- Added a pre-post condition documentation to the Board constructor in section 3.2.10
- Updated uses relationship to include AI class
- Updated test plans to include black box and white box testing to the requirements of assignment 3.
- Added description of the AI algorithm, and the implementation of the board
- Updated internal design review to reflect design decisions in assignment 3.
- Work distribution log, meeting minutes, and acknowledgements added in order to rectify any confusion as to which students contributed to each part of the document.

10.2 Work Distribution Log

The following table lists tasks accomplished by each member of group 18 for assignment 3.

Member	Tasks Accomplished
Zichen Jiang	• Wrote all of the code for the AI • Wrote all of the JavaDocs/comments for the AI class • Drafted the AI algorithm explanation for the documentation
Danish Khan	• Updated all of the MIS/MID • Made tabular expressions for the MIS/MID • Made pre/post conditions for the MIS/MID
Kelvin Lin	• Wrote all of the test plans/test cases • Revised the uses relationship, and the internal review • Wrote and edited the AI algorithm explanation based on Zichen's draft
Hasan Siddiqui	• Wrote code for the MenuView

10.3 Meeting Minutes

The following section contains the archive of meeting minutes recorded by group 18 since the beginning of assignment 1.

10.3.1 Feb 1, 2016

Attendance: Zichen Jiang, Hasan Siddiqui.

10.3.1.1 What we discussed:

- We have went through the document to figure out the requirements of this assignment.
- We decided the timeline of the assignment as follows:
- Documentation/requirements done by Feb 10th (preferably), and no later than Feb 12th.
- Coding/JUnit testing done by Feb 18th
- Start reviewing/test report/code documentation done by Feb 21st

10.3.1.2 Questions:

4.2 description of the semantics vs. description of the syntax
Slides 3 MIS MID
4.3 view of the uses relationship
Tutorial 4 slides
4.4. include a trace back to requirements in each class interface;
Traceability

10.3.1.3 Next meeting:

Feb 2nd, determine another meeting time for the week
Meeting after that (Wednesday):

- bring your ideas for implementation and decide on a best implementation
- determine the classes and their names

10.3.1.4 Meeting after that (Friday):

- discuss detail of the classes, i.e., their methods and variables
- start writing documents (4.1 - 4.5)

10.3.1.5 Schedule:

3:00 pm, Thode. Feb. 3. 2016

10.3.2 Feb 2, 2016

Attendance: Zichen Jiang, Danish, Hasan

10.3.2.1 What we discussed:

- We have finished discussing abstract classes
- We have started on making fields and methods for each class

10.3.2.2 Questions:

- highlight error, or just display error?

10.3.2.3 Next Meeting:

Friday Feb. 5th 6-8PM

10.3.3 Feb 5th, 2016

Attendance: Zichen Jiang, Danish, Hasan

10.3.3.1 What we discussed:

We have finished a prototype consist all the classes and their public and private entities in UML.

10.3.3.2 What's next:

Zichen will complete 4.1 to 4.5 of Assignment 1 by Feb 10 or 12 and upload it to Github. Danish and Hasan will do an internal review (4.6) on the document. Then Danish and Hasan will start coding; during which, they will complete 4.7. Danish and Hasan will also test each others' code and write up the test document. The coding and testing should be done by Feb 18.

There will be another meeting on Feb 17. The exact time is TBD.

10.3.4 Mar 9th, 2016

Attendance: Zichen, Danish, Hasan

10.3.4.1 What we discussed:

We went over all the code to make sure everyone understands what each part of the program does. We have assigned work for everyone.

10.3.4.2 What's next:

Zichen will rework on the document, more specifically, MIS, Uses, and some other parts. Hasan will work on the win condition determination, and complete the MID part of his code. Danish will work on the load and save functions, and complete the MID part of his code.

10.3.5 March 14th

Attendance: Kelvin, Zichen, Danish, Hasan

10.3.5.1 What we discussed

Kelvin showed and explained his work for assignment one from his group. The others observed his code to understand it. We have discussed two possibility for assignment two:

1. Use Kelvin's work. It only needs a few addition to complete assignment two. However, we have to digest all of his code and documentation. There is a high transition cost.
2. Continue with our work. We will have to spend some time to fix things we did wrong in the first assignment. And it would take longer to code for assignment two. But the training cost would be lower, since there is only one person to train.

We have not yet decided which direction we want to go.

10.3.5.2 What's next

Hasan will review Kelvin's code and see how long it would take to separate the View and Controller in our original work. We will make a decision by Tuesday midnight, and start from there.

10.3.6 March 17th

Attendance: Kelvin, Zichen, Danish
(Hasan is sick)

10.3.6.1 What we discussed

We have decided that to use Kelvin's work. We also went over the requirement for assignment two.

10.3.6.2 What's next

Kelvin is going to finish the code for assignment two. He will also complete the private implementation part of documentation. In the meanwhile, Zichen, Danish, and Hasan will fix all the errors from assignment one (documentation wise). After that, we will work together to document the changes Kelvin made.

10.3.7 March 20th

Attendance: Kelvin, Danish, Zichen
(Hasan is at home)

10.3.7.1 What we discussed

The game needs to be tested.

10.3.7.2 What's next

Item	Worker	Status	Merged into main doc?
Add MIS/MID for each module. (5.1, 5.2, 5.4)	Danish (Board Controller MIS/MID) Alic (Board) Kelvin (Views)	Done	yes
Expand on Uses Relationship (5.3)	Kelvin	Done	yes
Update internal review (5.5)	Kelvin	Done	yes
Write comments for the code (5.6)	Kelvin	Done	n/a
Add a changelog appendix to indicate what additions we made to the report (5.7)	Kelvin		
Test report (6)	Hasan		yes
Formatting	Alic	In progress	n/a
Private implementation	Alic	Done	yes
Traceability	Alic	Done	yes
Decomposition (justify the decomposition by using the requirements for the assignment)	Kelvin	Done	yes

10.3.8 March 31, 2016

Attendance: Kelvin, Alic, Danish

10.3.8.1 What we discussed:

The following members will be responsible for the listed tasks below for assignment 3.
Alic: AI - 3, 4.7, 4.6
Hasan: UI – 1.2, 4.7
Danish: MIS/MID - 4.1, 4.2, 4.4
Kelvin: Testing, 4.3, 5, 4.5

All - 4.8 (record any changes in change log)

10.3.8.2 What's next:

Let's finish this!

10.4 Acknowledgements

Before the beginning of assignment 2, member Kelvin Lin was part of group 16 along with Jeremy Klotz and Kerala Brendon. However, due to undisclosed reasons, the members of group 16 were regrouped.

Kelvin was regrouped to group 18, and group 18 was given the option of using either their existing assignment 1, or group 16's assignment 1 as the basis for assignment 2 and 3. Group 18 chose to use group 16's assignment 1, including the documentation.

Therefore, portions of this report reflective of assignment 1 were created by the following members:

1. Kelvin Lin – linkk4 – 1401464
2. Kerala Brendon – brendokh – 1424625
3. Jeremy Klotz – klotzj – 1426853

10.5 List of Tables

10.6 List of Figures