# Assignment 1

Kelvin Lin, Kerala Brendon, Jeremy Klotz
Student Number: 001401464, 001424625, 001426853
Course Code: SFWR ENG 2AA4/Comp Sci 2ME3

February 22$^{\text{nd}}$, 2016

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

# Contents

# 1 Introduction and Architecture

This report will document the design and the decisions made in Assignment 1 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-Controller-View (MCV) architecture. The MCV architecture was chosen because it accentuates the principle of separation of concerns. The MCV architecture consists of three main components: the model, the controller, and the view. With the MCV 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.

# 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 separations 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):

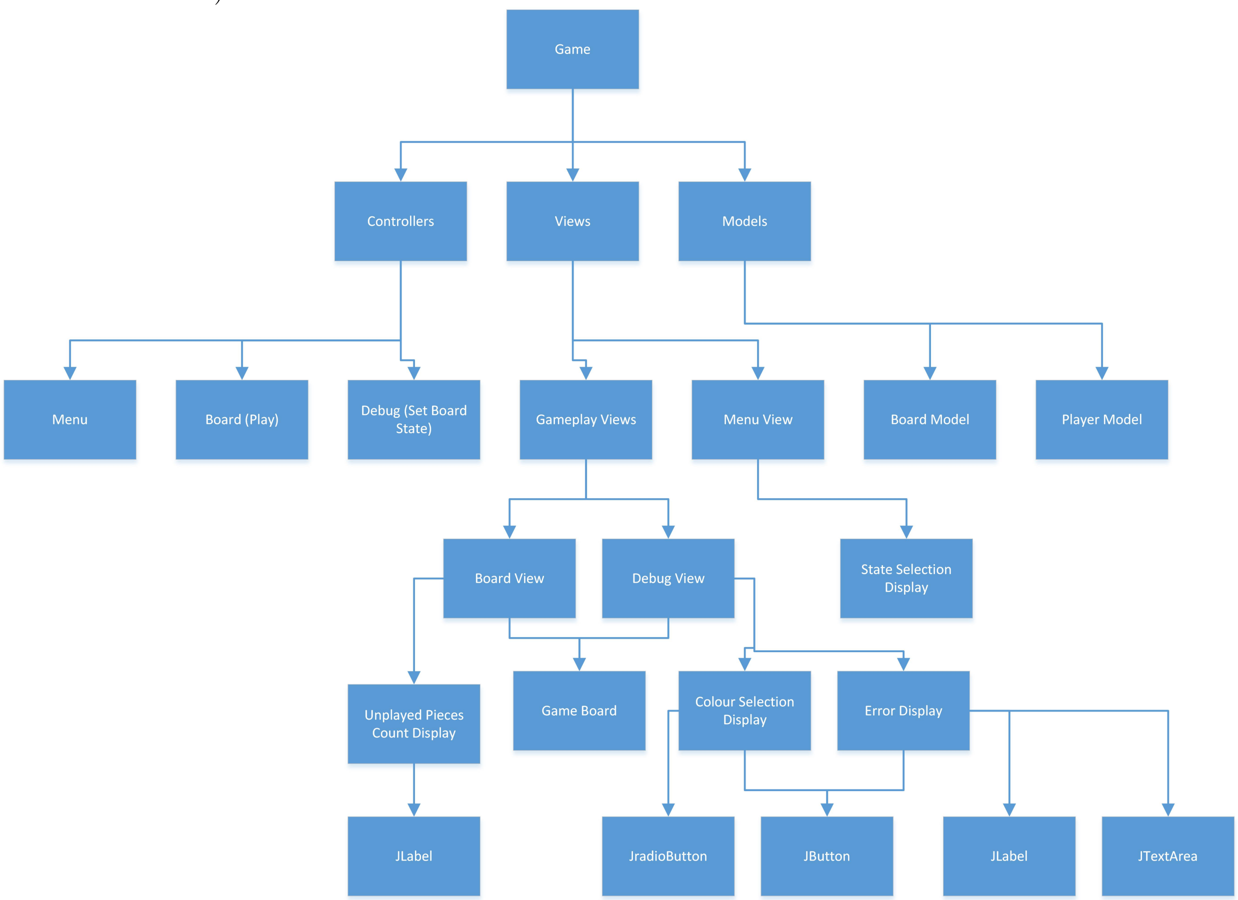


Figure 1: The Decomposition Hierarchy

# 3 Module Guide

## 3.1 MIS

**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

**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.

**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

**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.

**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.

**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

**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.

**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, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

**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 $(x_{topleft}, y_{bottomright})$

**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

**CLASS: VIEW**

What the user sees...

**INTERFACE**

**USES**

Board

Circle

**TYPE**

None

**ACCESS PROGRAMS**

**View(int N)**

Constructor method of type View. Creates a board of the current application state.

**setBoardState(int number, int state): void**

Set the state of the board.

**getBoardState(int number): int**

Return the current state of the board from accessing getPieceState.

**getCircles(): Circle[]**

Return the array of all circles in the board.

**setStates(int[] states): void**

Set the states of the game.

**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)

**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

**TYPE**

none

**ACCESS PROGRAMS**

**BoardController(int N)**

Construct the board.

**BoardController(int N, int[] boardState)**

Construct or update the board based on the correct state

**CLASS: BoardView**

**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 EventListers needed to obtain input from the user.

**pieceNotTaken(int number): boolean**

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

**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.

**Class: Game**

**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.

### 3.2 MID

**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

radius: double
The radius of the circle.

**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;

**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

boardView: BoardView

NUMBER_OF_PIECES = 6: int
Instantiate number of pieces per player, if this was 9 men's morris, it would change to 9 etc.

BLUE_STATE = 1: int
Blue state is index 1

RED_STATE = 2: int
Red state is index 2

FONT_SIZE = 25: int
Set default font size

DEFAULT_SCREEN_WIDTH = 500: int
Set default screen width

DEFAULT_SCREEN_HEIGHT = 500: int
Set default screen height

blue, red, black: JRadioButton
Series of JRadio buttons

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

playGame: JButton
Will change the application state to regular gameplay

N: int
Number of layers/ rectangles

**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.

| | | | | |
|---|---|---|---|---|
| blueCount<= NUMBER_OF_PIECES | redCount<= NUMBER_OF_PIECES | blueCount <=1 | redCount<3 | Legal |
| | | | redCount>=3 | Illegal |
| | | redCount <=1 | blueCount<3 | Legal |
| | | | blueCount>=3 | Illegal |
| | | blueCount>1 | | Legal |
| | | redCount>1 | | Legal |
| | redCount>NUMBER_OF_PIECES | | | Illegal |
| blueCount>NUMBER_OF_PIECES | | | | Illegal |

**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
mouseEntered(MouseEvent e): void
mouseExited(MouseEvent e): void
mousePressed(MouseEvent e): void
mouseReleased(MouseEvent e): void

**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.

**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.

**CLASS: MENUCONTROLLER**

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

**IMPLEMENTATION**

**USES**

JFrame
MenuView

**VARIABLES**

jFrame: JFrame
view: MenuView
nextState: int

**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 |

**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

playGame: JButton
User will click to go directle to play state

debug: JButton
User will click to go to debugging and then to play state

state: int
Keeps track of the state the game is in

defaultFontSize = 36: int
Sets the default font size

defaultScreenWidth = 500: int
Sets the default screen width

N: int
The number of layers.

**ACCESS PROGRAMS**

**MenuView(int N)**

Constructor method.
Instantiate play game and debug buttons
mouseClicked method
Instantiate Box

**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.

**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.

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

**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

**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
**ACCESSOR 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, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

**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

**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, $w = |x_{topleft} - x_{topright}|$. This will be used for assistance in properly scaling based on window size.

**getIntHeight():int**

Return the height of the rectangle, $h = y_{topright} - y_{bottomright}|$. 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 $(x_{topleft}, y_{bottomright})$

**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**

**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.

NUM_PIECES_PER_LAYER = 8: int

Amount of pieces per layer

pieces: int[]

Amount of positions to place pieces.

**ACCESS PROGRAMS**

**Board(int N)**

Determine the number of pieces
pieces = N*NUM_PIECES_PER_LAYER

**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.

**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

**VARIABLES**

jFrame: JFrame

boardView: BoardView

turn: int
If turn is 0 then it's blue's turn, if it's 1 then red's turn.

blue, red: Player
state = 0: int
If it is state 0, place pieces. If it is state 1, play game
NUMBER_OF_PIECES = 6: int
Number of pieces we're using. This can change to 9 if we are going to do 9 Men's Morris instead.

BLUE_STATE = 1: int
Blue state has a value of 1

RED_STATE = 2: int
Red state has a value of 2
FONT_SIZE = 25: int
Declaring a size for the font used in the application
DEFAULT_SCREEN_WIDTH = 500: int
The default width of screen (will scale if stretch/compress window)

DEFAULT_SCREEN_HEIGHT = 500: int
The default height of screen (will scale if stretch/compress window)

blueLabel, blueCount, redLabel, redCount, cLabel, cCount: JLabel
Some labels to properly update the view

**ACCESS PROGRAMS**

**BoardController(int N)**

Framework for the game, manipulates the model to know how to update the view.
Instantiate Random Turns
Instantiate Models

Instantiate Views

**BoardController(int N, int[] boardState)**
Construct or update the board based on the correct state. Add up all 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

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

**updateLabels(): void**
This method will update the labels of each player involved.

**updateView(): void**
Controller takes information from the view and calls methods from the java.awt library
on it. Repaints if it has been changed.

**MouseClickEventHandler implements MouseListener**
Contains all possible methods that may be used for the Six Men's Morris Game. If we decide to use other MouseListener methods, we can slightly change the code to do so.
**mouseClicked(MouseEvent e): void**
This method allows for alternate colour pieces to be placed on the board after each click.

```
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 blue number of unplayed pieces > 0
                                        set that spot at i as 2 (red)
                                        place blue piece
                                increment turn
                case 1:
                        if pieceNotTaken at i on boardView
                                if red number of unplayed pieces > 0
                                        set that spot at i as 2 (blue)
                                        place red piece
```

decrement turn
update labels and view
if red and blue number of unplayed pieces == 0
state =1
else if 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 |

mouseEntered(MouseEvent e): void
mouseExited(MouseEvent e): void
mousePressed(MouseEvent e): void
mouseReleased(MouseEvent e): void

**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

N: int
Number of squares

states: int[]
Array of integers that holds each state

COLORS = { Color.BLACK, Color.BLUE, Color.RED }: Color[]
If a third colour was introduced, add it here

RECTANGLE_WIDTH_SCALING = 0.19: double
Rectangle width scaling

RECTANGLE_HEIGHT_SCALING = 0.19: double
Rectangle height scaling

CIRCLE_SCALING = 0.07: double

Circle scaling

HORIZONTAL_LINE_SCALING = (CIRCLE_SCALING/RECTANGLE_WIDTH_SCALING)*0.9: double
Horizontal line scaling

VERTICAL_LINE_SCALING = (CIRCLE_SCALING/RECTANGLE_HEIGHT_SCALING)*0.9: double
Vertical line scaling

circles: Circle[]
Array of circles , will be used multiple times
**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
Draw 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.

**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.

# 4 Trace to Requirements

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

| **Requirements** | **Modules** |
|---|---|
| Enable the user to set up a board to play the game. | BoardBoard, Controller, BoardView |
| The board includes two types of discs. | BoardController, BoardView |
| The discs are placed on either side of the board. | BoardController |
| There are no discs at the start of the game. | BoardController, Board, BoardView |
| The order of play is determined randomly. | BoardController |
| The user should be able to start a new game, or enter discs to represent the current state of a game. | MenuController, MenuView |
| 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 |
| When all the discs the user wants to play have been played, the system should analyze whether the current state is possible. | DebugController |
| Errors should be displayed to the user. | ErrorDialog |

Table 1: Trace to Requirements

# 5 Uses Relationship

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

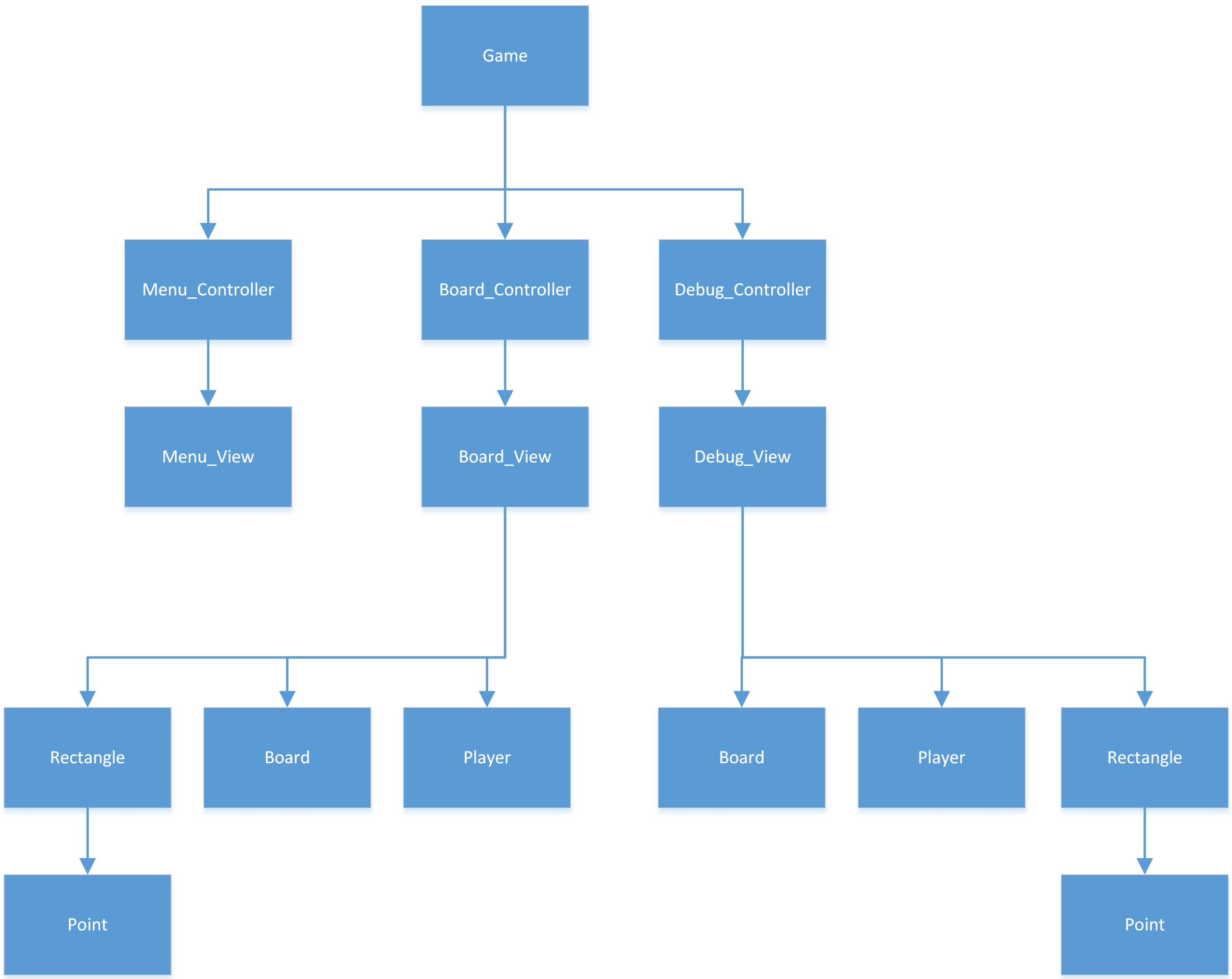


Figure 2: The Decomposition Hierarchy

# 6 Anticipated Changes & Discussion

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

- The Board_Controller 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 Board_View and the Debug_View
- 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.

## 6.1 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-Mens Morris can be implemented as another module within the same application.

## 6.2 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.

## 6.3 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.

## 6.4 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.

### 6.5 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.

### 6.6 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 $2^{32}$ moves.

### 6.7 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.

### 6.8 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.

## 7 Test Plan/Design

### 7.1 Requirement 1: Enable the user to set up a board to play the game

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

Table 2: Testing Requirement 1

| **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. |

### 7.2 Requirement 2: The board includes 2 types of discs

Table 3: Testing Requirement 2

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

Conclusion: There are red and blue discs

### 7.3 Requirement 3: The discs are placed on either side of the board

Table 4: Testing Requirement 3

| **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. |

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.

### 7.4 Requirement 4: There are no discs at the start of the game

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

Table 5: Testing Requirement 4

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

### 7.5 Requirement 5: The order of play is determined randomly

Table 6: Testing Requirement 5

| **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 |

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

### 7.6 Requirement 6: The user should be able to start a new game, or enter discs to represent the current state of the game

Table 7: Testing Requirement 6

| **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. |

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

## 7.7 Requirement 7: 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

Table 8: Testing Requirement 7

| 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. |

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.

## 7.8 Requirement 8: When all discs the user wants to play have been played, the system should analyse whether the current state is possible

Table 9: Testing Requirement 8

| 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. |

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.

### 7.9 Requirement 9: Errors should be displayed to the user

Table 10: Testing Requirement 9

| Input | Result |
|---|---|
| Run MenuController, choose Debug, place 10 blue pieces and 6 red pieces. Play game. | Error window appears, there are too many pieces. |
| Press OK on error message, replace all pieces with black. Play game | Error window appears, both players have fewer than 3 pieces. |

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 Conclusion

This report details the design decisions made and the implementation of assignment 1. 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 an 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 MCV architecture. The components of the MCV 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 can be easily modified to accommodate future changes. Both separation of concerns and modularity in our code allow us to implement future changes with minimal change to the overall interface of the code.

Therefore, since our implementation of assignment 1 embodies all of the principles of software engineering, we believe that our application is a robust implementation of Six Men's Morris.