Hatim Rehman | Louis Bursey | Sarthak Desai

SIX MEN's MORRIS

2AA4 Assignment 3

TABLE OF CONTENTS

Text with this highlight indicates additions to previous report.

Text such as this is unchanged work from previous report.

~~Finally, text in this style indicates removal of content from previous report~~.

Some Assumptions for Design of Game and Intentional Behavior:

1. Transitioning from Setup Scenario to starting the game assumes neither players have new pieces remaining outside of the board to place – otherwise they would have been placed already.
2. A player may not win the game by trapping their opponent's pieces. This will result in an automatic stalemate. This follows the standards of other board games like Chess.
3. The orange rectangle:
   - In Setup Scenario: Starts the game.
   - When game has started: Saves the game.
4. This game is designed and coded to be **dynamic**. Changing one field (MORRISSIZE) in the Controller class changes the layout from 6 men's morris to 9 men's morris, or any other logically equivalent variant. Not tested for 9 Men's Morris).
5. Players can only place 6 pieces maximum (assuming 6 men's morris, this is a dynamic field that changes with the MORRISSIZE variable, along with every other piece of the game). A dynamic counter is stored in their class of remaining pieces.
6. Players **can** place pieces on top of other placed pieces in the setup scenario. This effectively returns the previous piece to the old player (since this is a setup mode). This allows for mistakes in piece placement.
7. There is no delay between when a user places a piece and the computer's turn.
8. When players are in the middle of placing pieces, and the user forms a mill, sometimes it may seem as if removing the computer's piece may "not do anything". What is happening is the user has removed the piece, and the computer has placed its next piece in that position right away.
9. It was asked of us to use the same method to determine if AI goes first, however, a new method was created called computerGoesFirst(). This is because the previously implemented game only starts when a mouse click happens, so in the situation the computer was to go first a different method was needed to make the computer's move prior to starting the game, or have the player always start first. The method then returns a boolean that Controller.java uses to determine what happened.

4.1 - Description of Classes/Modules

The layout of the classes and modules in this project follow the MVC structure as follows:

	Class	Description
Controller	Controller.java	Main controller class of the application. This class creates all the objects necessary to the game (Ellipse and Player), and handles the View. Input from the View is also received here, and changes to the Model are made if necessary and the View is reconstructed. This class uses the gameFunctions class to determine the state of the game and control the AI.
	gameFunctions.java	Contains all methods necessary for controlling the flow of the game and deciding the moves of the AI. Some methods include checkMill(...), checkRemove(...), switchTurn(...), saveGame(...), etc.
*Mode*l	Ellipse.java	Child class to java.awt.geom.Ellipse2D.double. Objects of this class represent positions on the board.
	Player.java	Child class of Ellipse.java. Objects of this class represent the Players of the game.
View	updateBoard.java	Creates a board based on the state of the Model objects. This class is called every time a change is made to the state of the Model objects by the Controller class.
	mainScreen.java	Creates the first screen the players see when the game is launched.

Decomposition Analysis: There are now two Controller classes to improve modularity. Controller.java is in essence the class that is responsible for the flow of the game (i.e. what are the consequences of the previous move, and what should happen next), whereas gameFunctions.java holds all methods that allows Controller.java to proceed with user input, and decide the moves of the AI. There is need to split the Model into more than one class because the objects needed have unique attributes. Similarly, there are two unrelated screens that the user sees – the initial screen that allows the players to pick a game mode, and the screen displaying the board where the actual game is played, creating the need to have more than one class for the View.

4.2 Public Interface

CLASS: Controller.java

Main controller class of the game that instantiates all objects and classes, and controls the flow of the game. All user input is processed here.

USES:

gameFunctions.java
mainScreen.java
updateBoard.java
Ellipse.java
Player.java

VARIABLES:

WIDTH: int
Stores the width of the window.

HEIGHT: int
Stores the height of the window.

MORRISSIZE: int
Stores which board to load (preferably six men's morris or nine men's morris).

SCALE: int
Scales the window.

diskHolder: Ellipse[][]
Holds all the objects representing locations on the board.

player1: Player
Object representing player 1.

player2: Player
Object representing player 2.

computer: Player

Object representing the AI.

END: Rectangle

Object representing the end or save button.

boardColor: Color

Color of the board.

state: int

Represents which state the current game is currently in.

lastState: int

Represents the previous state that the game was in prior to current state.

displayState: String

String variable that displays the status of the game (i.e win/loss/in progress).

sameTurn: boolean

Boolean value used by updateBoard to not update turn if a mill was formed.

i_old: int

Previously private, represents which square the piece to be moved was in.

j_old: int

Previously private, represents which cell the piece to be moved was in.

againstComputer: boolean

Boolean that is set to true if user clicks on game mode against AI.

computersTurn: boolean

Boolean that is set to true whenever the computer must play next.

xCoord: int

X coordinates of the piece on the board that is clicked or referred to by the AI.

yCoord: int

Y coordinates of the piece on the board that is clicked or referred to by the AI.

ACCESS PROGRAMS:

Controller( ):
Behavior: Instantiates Ellipse objects for every position on the board and places them in array, and instantiates two Player objects.
How: for-loop that iterates over the *MORRISSIZE* field. Predetermined position for each Ellipse object that changes by a constant (x,y) factor in every iteration. Player objects created afterwards.

startGame(String startCondition): void
Behavior: Can start a new game, load a previous saved game, or launch a game from scenario setup. Controls the state of the game.
How: Uses startCondition to launch the correct game scenario. Controls the state based on helper methods in gameFunctions, and mouse input from the view classes.

scenarioSetup( ): void
Behavior: Starts the scenario setup mode and changes the state of the objects depending on user input.
How: Modifies the state variables in the diskHolder[][] array wherever the user wants to place a piece. Keeps track of legality of user action using conditional statements and accessing state variables from the relevant objects.

CLASS: gameFunctions.java

Second Controller class that holds ALL helper functions that the main controller class needs. Some examples of the helper methods would be switching the turn, saving the current state of the board, or checking the legal moves for the currently selected piece, etc.

USES:

Ellipse.java
Player.java

VARIABLES:

None in the public interface.

ACCESS PROGRAMS:

checkRemove(int square, int cell): boolean:
Behavior: Returns a boolean indicating if piece in location [square][cell] can be removed.
How: Conditions for returning true are if the piece is the opponent's piece, and is not part of a mill (unless only mill and player has no more pieces to place).

getEmpty(int square, int cell): ArrayList<int[]>
Behavior: Returns an ArrayList that holds [i][j] locations that a given piece [square][cell] can move to.
How: Modulo arithmetic on the piece to determine all available locations based on the setup of diskHolder[][] array and how the locations are stored.

switchTurn(Color a): Color
Behavior: Returns the color of player that should follow *a.* Decrements *a*'s piece counter.
How: Determines which player is *a* and *not a*. Then call *a.decrementRemaining( )*, and return (*not a*).getColor( ). Pre/Post Condition showing this function is as follows:

{ i = player1.getColor() V o = player2.getColor() }
P
{ o = player1.getColor() V o = player2.getColor() } Λ { o != i }

turnRand( ): Player

Behavior: Returns randomly which Player should go first.

How: Uses Math.random() and arbitrarily assigns one half of the range to each player. Whichever player is closer to the number goes first.

checkMill(int square, int cell): int

Behavior: Returns the number of mills formed at position [square][cell]. Will be called after every new piece is placed.

How: Shown below (d[][] is diskHolder array and mills is always 0 at start) :

<table>
<tr><th colspan="3">Condition</th><th>Result</th></tr>
<tr><td rowspan="2">cell % 2 == 1</td><td colspan="2">d[square][cell + 1] == (color) AND d[square][(cell + 2) % 8] == (color)</td><td>mills++</td></tr>
<tr><td colspan="2">NOT d[square][cell + 1] == (color) OR NOT d[square][(cell + 2) % 8] == (color)</td><td>NC</td></tr>
<tr><td rowspan="4">cell % 2 != 1</td><td rowspan="2">d[square][cell + 1] == (color)
AND d[square][(cell + 2) % 8] == (color)</td><td>d[square][(cell + 7) % 8] == (color)
AND d[square][(cell + 6) % 8] == (color)</td><td>mills += 2</td></tr>
<tr><td>NOT d[square][(cell + 7) % 8] == (color)
OR NOT d[square][(cell + 6) % 8] == (color)</td><td>mills++</td></tr>
<tr><td rowspan="2">NOT d[square][cell + 1] == (color)
OR NOT d[square][(cell + 2) % 8] == (color)</td><td>d[square][(cell + 7) % 8] == (color)
AND d[square][(cell + 6) % 8] == (color)</td><td>mills++</td></tr>
<tr><td>NOT d[square][(cell + 7) % 8] == (color)
OR NOT d[square][(cell + 6) % 8] == (color)</td><td>NC</td></tr>
</table>

Table: Parnas table showing simple mill checker function.

checkBoard( ): String

Behavior: Returns a String message that determines the legality of the board.

How: Runs through all the cases of an illegal board state and returns all errors in a String.

gameState( ): String

Behavior: Returns a string on the current state of the game.

How: Uses private methods checkLoss( ) and checkDraw( ) to determine state.

saveGame(String filename): void

Behavior: Saves the current state of the game in a file called *filename*.txt.

How: Uses BufferedWriter to write the content of the current Turn, diskHolder's [i][j]'th object's color, the remaining pieces of each player, the values of state and displayState, and boolean value of againstComputer onto a text file called *filename*.txt

loadGame(String filename): boolean
Behavior: Loads the state of the game saved in *filename.*txt.
How: Uses BufferedReader to set the states of the variables Turn, diskHolder's [i][j]'th object's color, the remaining pieces of each player, the values of state and displayState, and boolean value of againstComputer using the values written in the text file *filename.txt*. If againstComputer was true, it initiates the computer object as the player that is not supposed to go next (since computer will not save the game on its turn).

computerGoesFirst(): boolean:
Behavior: Determines randomly if AI should go first, and if so makes a move for the AI. A separate method is used because to get into the switch statements a mouse click is needed, so if computer goes first its turn must be completed prior.
How: Uses math.random(), then calls getOptimal(Player a).

getOptimalMove(Player a): void
Behavior: Determines optimal location the computer can place a piece. Optimality conditions equal whether a mill can be formed or blocked. Otherwise default to random.
How: 3 for loops, one for determining if a mill can be formed or blocked using overloaded checkMill(...) method for both player colors. The other for loop tries to extend mills, determined by modulo arithmetic on data structure. Third loop for random location.

getOptimalMove(Player a, Player b): void
Behavior: Determines the first piece which piece *a* can remove from *b*.
How: Goes through every piece on the board in nested for loop looking for a piece that matches *b.getColor().* Returns first one.

getOptimalMove(Player a): ArrayList<int[]>
Behavior: Determines randomly which piece belonging to *a* can be moved.
How: In a nested for loop, first goes through each piece whose color matches *a.getColor()* and checkRemove() returns a non empty ArrayList for, counting each occurrence. Then using the Random library, determines a piece from these pieces to move and goes

through the nested for loop until this piece is found, and calls the private getOptimalMove() which will also move it to a random location from possible locations.

CLASS: updateBoard.java

View class that is responsible for displaying the current state of the board using the state variables in the Ellipse objects in diskHolder[][] array, and the state variables in the Player objects.

USES:

Ellipse.java
Player.java

VARIABLES:

None in the public interface.

ACCESS PROGRAMS:

paintComponent(Graphics g): void

Behavior: Displays the board using current state of each position.

How: *g.*setColor(object_that_goes_here.getColor());
g.fill(object_that_goes_here);

getPreferredSize( ): Dimension

Behavior: Scales the graphics appropriately

How: Uses HEIGHT and WIDTH variables as scaling factor.

CLASS: mainScreen.java

View class that displays the first screen that the players see. This page allows users to select a new game, launch a custom game (Scenario Setup), or load a game from the last save file (if one exists).

USES:

None

VARIABLES:

None in the public interface.

ACCESS PROGRAMS:

mainScreen( ): void

Behavior: Creates label. Instantiates buttons.

How: Calls setter methods. Creates new Label object for label.

getButton1( ) : JButton
Behavior: returns “New Game” button.
How: return button1;

getButton2( ) : JButton

Behavior: returns “Scenario Setup” button.

How: return button2;

getButton3( ) : JButton

Behavior: returns “Load from last save” button.

How: return button3;

getButton4( ) : JButton

Behavior: returns “Play against Computer” button.

How: return button4;

getPreferredSize( ): Dimension

Behavior: Scales the graphics appropriately

How: Uses HEIGHT and WIDTH variables as scaling factor.

CLASS: Ellipse.java

Child class to Java's Ellipse2D class that is modified to include methods that store the Color object. Objects of this class represent locations on the board.

USES:

None

VARIABLES:

None in the public interface.

ACCESS PROGRAMS:

Ellipse(double a, double b, double c, double d)

Behavior: Sets fields from parent class, and sets default color to be boardColor.

How: Super(...)

setColor(Color color): void

Behavior: Sets color field.

How: this.color = color;

getColor( ) : Color

Behavior: Returns color.

How: return color;

CLASS: Player.java

Child class to Ellipse.java class that is modified to include counter methods for remaining pieces. Objects of this class represent the Player(s).

USES:

None

VARIABLES:

None in the public interface.

ACCESS PROGRAMS:

Player(double a, double b, double c, double d)

Behavior: Sets fields from parent class)

How: Super(...)

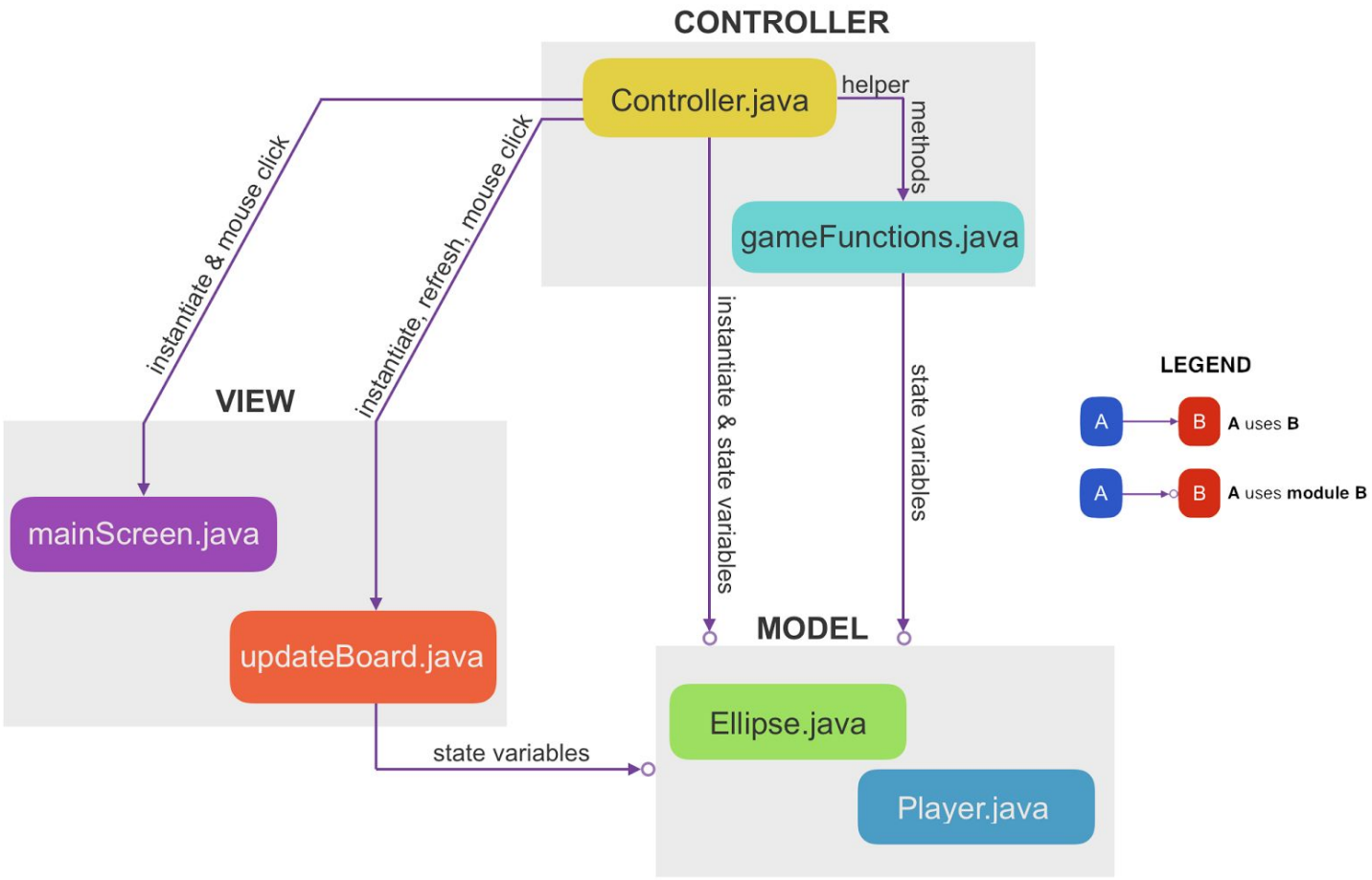
getRemaining( ): int
Behavior: returns remaining field.
How: return remaining;

decrementRemaining( ): void
Behavior: Decrements remaining pieces.
How: remaining--;

restoreRemaining( ): void
Behavior: Increments remaining pieces.
How: remaining++;

setRemaining(int x): void
Behavior: Sets custom value to remaining.
How: remaining = x;

4.3 Uses Relationship



4.4 Requirements Traceback

Requirements	Class/Method
Board should include two different kinds of discs	Class: Player (Two instances instantiated in the Controller class during constructor method)
The order of play (blue first or red first) shall be determined randomly	Class: Controller Method: turnRand()

The user shall be able to choose to start a new game, or enter discs to represent the current state of a game by placing different coloured discs in the frame.	Class: mainScreen (view choose a game mode) Class: updateBoard (view for scenario setup) Class: Controller Method: startGame() (controls the input of the user)
Allow the user to select a colour and then click on the position for that disc.	Class: Controller, Method startGame() Class: Ellipse (update color on board) Class: Player (update player counter)
The end of the setup scenario should be indicated by the user in some way	Class: Controller. Method checkBoard() (displays validity of the state of the board in a pop up - pass or fail).
Analyze whether the current state is possible or not	Class: Controller, Method checkBoard() (same as before).
All the errors shall be highlighted	Class: Controller, Method: checkBoard()
The application has to determine if the game has been Won, Drawn or is in Progress	Class: gameFunctions, Method: checkloss(), checkdraw(), gamestate()
The result must be displayed at all times	Class: updateBoard (the paintComponent method displays a string that shows the state of the game

	as determined by the gameFunctions class)
Must be able to make legal moves	Class: gameFunctions Method: getEmpty() (gets all empty positions that a piece can move to)
Must be able to make moves in turn	Class: gameFunctions Method: turnRand() (determines who goes first), switchTurn() (switches turn based on who just went)
User should be able to start a new game, save game and load an existing game.	Class: gameFunctions, mainScreen Method: saveGame(), loadGame() getButton3() (button in view that upon click will direct to loadGame method
Able to remove pieces	Class gameFunctions, Controller Method: checkRemove(), startGame()
Game ends when a player has won	Class: gameFunctions, Controller Method: checkBoard(), checkLoss(), checkDraw();
Choose between two modes of operation: player v player or player v AI	Class: gameFunctions, mainScreen Method: startGame() (depending on parameter, will load the right mode) getButton4() (button in view that upon click will direct to mode against AI.

Decide if the computer plays first.	Class: gameFunctions Method: computerGoesFirst()
All rules for player v AI are the same as player v player	Class: gameFunctions (All the same legality checks are in place for the AI, in fact very little to none of that code was touched).
Algorithm to determine the next best legal move for the AI	Class: gameFunctions Methods: getOptimalMove() (overloaded method that differs by parameters and/or return type called appropriately from each state the game could be in).

4.5 Description of Private Implementation

Class	**Private Methods/Instance Fields**	**Description**
~~Controller~~	~~private static int turn~~	~~Variable to store which player's turn it is.~~ ~~Variable is used by in class methods to randomize a player's turn at the start of the game and to switch turns respectively.~~
Controller	private static JFrame frame	Variable to define the basic shell of the window for the game screen. Is used by in class methods to define the aspects about the game window. This includes dimensions, title, background colors, window resizing and visibility.
Controller	private static JPanel board	Variable to add game components to the frame. Is used in class by methods to add graphical representation of the board, including player pieces and buttons for the game's main menu to the JFrame.

Controller	Private static int mills	Variable to hold how many mills have been formed by the player. A player can form more than 1 mill at once. Is used by methods to assign the number of mills that are formed by a player once the board's status is checked. Methods manipulate the counter by decrementing it as a piece from the opponent is removed.
~~Controller~~	~~Private static int i_old~~	~~Variable to hold the ith index of the previous piece that was chosen before the current.~~ ~~Variable is used by methods to remember the location of the previous piece that was chosen in the case that the piece selected is trapped and cannot be moved. In this case a different piece needs to be selected~~
~~Controller~~	~~private static int j_old~~	~~Variable to hold the jth index of the previous piece that was chosen before the current.~~ ~~Variable is used by methods to remember the location of the previous piece that was chosen in~~

		~~the case that the piece selected is trapped and cannot be moved. In this case a different piece needs to be selected~~
Controller	private static ArrayList<int[]> legal	Variable to hold the legal locations that a selected piece can be moved to. Variable is used by methods to determine the locations that a piece can be moved on the board without creating a conflict. It is also checked to be empty, in which case the piece cannot be moved.
Controller	Private static boolean forceEntry	Variable to hold the boolean value depending on whether it is the AI's turn or not. Variable is used by the methods to determine if it is the AI's turn. If it is set to true then the program jumps straight to the case statements.
Controller	private static void createFrame	Method to generate the basic frame that is to be used for the view of the program.

		Creates a frame with the dimensions, background color and display settings that are specified in the program.
Player	private int remaining	Variable counter for the number of pieces that remain to be placed by a player on the board. Methods use the variable as a counter, decrementing it every time a piece is placed onto the board.
gameFunctions	private static final Color boardColor	Variable to hold the color of the board. Methods use the variable to check if there is a piece placed on a certain location on the board or not. If the color matches that of the board then there are no pieces
gameFunctions	private static Ellipse[][] diskHolder	Variable that is a 2-D array to hold all objects that are the pieces placed on the board. Methods use the variable to retrieve information about what piece is placed there and to be able to place and replace pieces on a location on the board.

gameFunctions	private static Player player1	Variable to hold the Blue player Methods use the variable to refer to the player's pieces, where they are on the board by comparing the colors, decrement the counter for a player's pieces and to determine who wins.
gameFunctions	private static Player player2	Variable to hold the Red player Methods use the variable to refer to the player's pieces, where they are on the board by comparing the colors, decrement the counter for a player's pieces and to determine who wins.
gameFunctions	private static int MORRISSIZE	Variable to determine the size of the board. That is if it is a six or nine men's Morris. Methods use the variable to know how many levels there are on the board. That is six men's morris would have 2 levels. But this can be changed to have 3 levels.
gameFunctions	private static String checkloss(Player x)	Method receives a player object as a parameter and checks if that player has lost the game or not. Returns a string that describes the state of the game.

gameFunctions	private static String checkdraw(Player x)	Method receives a player object as a parameter and checks if that player can make any moves. If not then the game is a draw. Returns a string that describes the state of the game.
gameFunctions	private static int checkMill(int square, int cell, Color color)	Method receives a square and cell value that marks a node on the board and a Color that indicates the player. Method checks how many mills have been formed (as there can be more than 1 mills formed). The method returns the number of mills formed.
gameFunctions	private static void getOptimalMove (ArrayList<int[]> legalMoves)	Method receives an ArrayList of moves that are legal to perform. The method then picks a location for a piece to move to.
Ellipse	private Color color	Variable to hold the color of the player whose turn it is. Is used by the View methods to color a position on the board with the last changed color (default white).
mainScreen	private static Jbutton button1	First button that is present on the main screen.

		Click on the button translates to the selection of a new game. In class methods use it display the text to allow the user view their choices
mainScreen	private static Jbutton button2	Second button that is present on the main screen. Click on the button translates to the selection of a Scenario Setup. In class methods use it display the text to allow the user view their choices
mainScreen	private static Jbutton button3	Third button that is present on the main screen. Click on the button translates to the selection of load previous game. In class methods use it display the text to allow the user view their choices
mainScreen	private static Jbutton button4	Fourth button that is present on the main screen. Click on the button translates to the selection of playing a game against the AI. In class methods use it display the text to allow the user view their choices

mainScreen	private JLabel Label1	Variable to hold the tile of the application. Class methods use variable to display the game's title to the screen
mainScreen	private JLabel Credits	Variable to hold the credits of the game developers Class methods use variable to display the names to the screen
mainScreen	Private static double HEIGHT	Variable defines the height dimension of the game frame that is presented to the user. Used by in class methods to return the height dimension upon method call.
mainScreen	private static double WIDTH	Variable defines the width dimension of the game frame that is presented to the user. Used by in class methods to return the width dimension upon method call.
updateBoard	private int MORRISSIZE	Variable to represent the size of the size of the board. This variable is used to determine what type of board is to be

		implemented. This can be edited to create a board for 6 or 9 men's morris
updateBoard	private double HEIGHT	Variable defines the height dimension of the game frame that is presented to the user. This is the fixed height of the frame. It is used by the in class methods to determine the dimensions of the board
updateBoard	private double WIDTH	Variable defines the height dimension of the game frame that is presented to the user. This is the fixed height of the frame. It is used by the in class methods to determine the dimensions of the board
updateBoard	private Player player1	Variable used to represent the player object for the first player This is used in the methods to retrieve the player's current state which includes the number of pieces that are left to play, what color does the player represent and to set the player color to the spots on the board.

updateBoard	private Player player2	Variable used to represent the player object for the second player This is used in the methods to retrieve the player's current state which includes the number of pieces that are left to play, what color does the player represent and to set the player color to the spots on the board.
updateBoard	private Ellipse[][] diskHolder	Variable that is a 2-D array that holds the state of the nodes on the game board. It is used by class methods to determine where on the board each player's pieces are.
updateBoard	private static Rectangle END	Variable to End the board frame.
updateBoard	private double SCALE	Variable to scale the board to fit different window sizes
updateBoard	private static final Color boardColor	Variable to hold the color of the board. Methods use the variable to set the color of the board when updating the board frame.

4.6 Design Evaluation

The logic of the game lies solely on the current state of the board - ie the color of every position. There are no actual unique objects for each player, just colors that represent their claim on the board. This means we are able to manipulate just one data structure, our 2D array containing reference to every position on the board, and do checks using patterns of colors. This, to us, was the optimal approach for the design of the project. One data structure is used for the view modules, and the same structure is used to validate game rules. It is the optimal approach, and we plan to stick with it for the future of the project.

AI ALGORITHMS:

The algorithms dictating the AI's behavior are simple for time's sake. When removing pieces and moving pieces, the AI looks for a legal move and makes it. When placing pieces, the AI first attemts to create or prevent a mill, then tries to put two of its pieces in a row to build a mill, then finally looks for any legal move. Further development on this project would go into research to improve the AI, but for the purposes of this assignment it is adequate.

Other design evaluations:

-The model we designed prior to implementing the project. It was successfully and accurately produced in the code.

–MVC structure has been more rigorously applied.

-The game is well designed for change, it can easily be adapted to a nine men's morris game, and the bones of the project can be kept for the next iterations. Making a dynamically constructed game was an objective, and it was met.

- The save game function is executed simplistically, using some kind of encryption would improve security. For the purpose of this assignment it is adequate.

Anticipated Changes:

AC1- n Men's Morris. The number of morrising men is largely decided dynamically, and can be changed by changing the secret in one class. A few checking methods will need an extra line to accommodate more squares, they are clearly labelled.

AC2- Play Against Computer. This will be accomplished by using another state in the main function. Functions to check if a piece is removable, check if a move makes a mill, etc. are modularized to make AI easier to implement.

AC3- The screen size is hidden, to make changing it easier.

In maintenence- Stronger AI logic

4.7 - Done in package

5.0 Testing Report

Black Box Testing

Test #: 1.0
Unit/System: Req2, order of play is determined randomly
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Started 20 different games	~10 blue and ten red plays first	11 blue first, 9 red	Pass

Test #: 2.0
Unit/System: Can create scenario, start new game, load game
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Tried all three buttons	Enter each mode	Entered each mode	Pass

Test #: 2.1.1
Unit/System: New game, able to make legal moves
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Moving pieces to adjacent square	Able to move if unoccupied	Able to move if unoccupied	Pass
Moving pieces to non-adjacent square	Unable to move	Unable to move	Pass
Moving pieces to occupied square	Unable to move	Unable to move	Pass

Test #: 2.1.2
Unit/System: New game, remove opponent's pieces when scoring a mill
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Score one mill, removing opp. non-mill piece	Can remove	Can remove	Pass
Score two mill, removing two opp. Non-mill piece.	Can remove twice	Can remove twice	Pass
Try to remove own piece/empty square	Can't remove	Can't remove	Pass
Try to remove from opp.'s mill, other opp. pieces not in mill	Can't remove	Can't remove	Pass
Trying to remove from opp.'s mill, opp only has mills	Can remove	Can remove	Pass

Test #: 2.1.3
Unit/System: Saving game
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
During moving	Can save, can load	Can save, can load	Pass

phase of turn, try to save	same game again	same game again	
During removing phase, try to save	Can't save	Can't save	Pass

Test #: 2.1.4
Unit/System: Game ends when necessary
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
One player has <3 pieces	Win message displayed, window closes	Win message displayed, window closes	Pass
Player has no moves on their turn	Draw displayed, window closes	Draw displayed, window closes	Pass

Test #: 2.2.1
Unit/System: Setup Scenario, starting to play
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Board state is legal, game started	Can play normally	Can play normally	Pass
Too few pieces, game started	Error message displayed	Error message displayed	Pass
Mill implies extra pieces	Error message displayed	Error message displayed	Pass

Test #: 2.3.1
Unit/System: Load game
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Loading old game	Board is the same as saved game, same functionality	Board is the same as saved game, same functionality	Pass

Loading with no game saved	New game started	New game started	Pass

Test #: 3.0
Unit/System: AI
Test Type: Black Box

Case	*Expected*	*Actual*	*Pass/Fail*
Starting game	Either player or computer starts first, computer moves if first	Computer started first, played legally	Pass
Playing against computer	Game plays normally, computer makes no illegal moves	Normal game w/ well behaved computer	Pass
Can save game vs computer	Saved game vs computer runs normally	Game saved, runs without error	Pass