2AA4 Assignment 2

Pavle Arezina 001410366 Group 19 Tutorial 2
Shady Nessim 001414948 Group 19 Tutorial 2
Alex Nguyen 001306407 Group 19 Tutorial 2
Abeed So'Jazyd Alibhai 001428809 Group 19 Tutorial 2

Contents

1 Introduction

Six Mens Morris was a popular board game in the Middle Ages till 1600s in Western Europe. Two players play on a board made up of an outer square and an inner square with 16 intersection points with each player receiving six pieces to place on the board. In our application the pieces will be coloured red and blue. The order of play will be determined randomly and pieces removed from the game are permanently removed. Players try to form 'mills, three of their own men lined horizontally or vertically, allowing a player to remove an opponent's piece from the game. A player wins by reducing the opponent to two pieces (where he could no longer form mills and thus be unable to win), or by leaving him without a legal move. The game is done in three phases. In the first phase of the game the players will place their pieces on the board in vacant spots, when this is accomplished the players will proceed to the second phase where the players move their pieces to adjacent vacant spots. When a player is left with three or lower pieces on the board, they may move their pieces to any vacant location. These movements are done in a turn by turn basis where each player moves one pieces and then allows their opponent to respond.

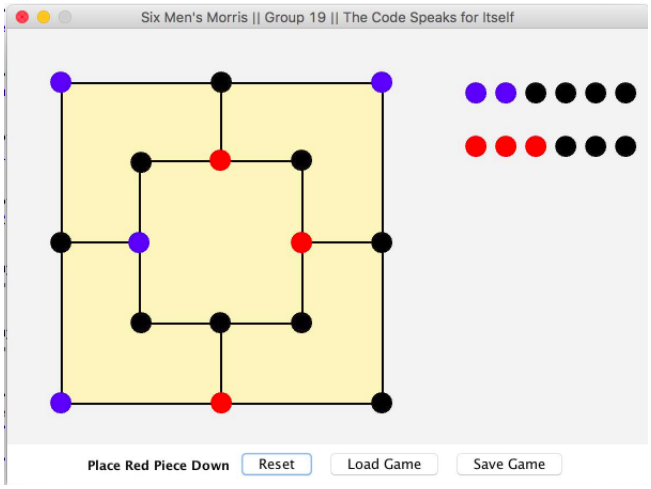

Figure 1: Six Men's Morris

1.1 Software

To implement the application of Six Mens Morris, the Java language will be utilized. This programming language has many advantages such as supporting object oriented programming and it is easy to use. The graphical portion of the application will be implemented with the standard libraries imported when installing JDK, mainly the project will utilize Java Swing, which is part of the Java Foundation Classes. This will allow any user with the latest Java SE

platform to be able to utilize the application.

1.2 Architecture

The Java application will be created with the Model-View-Controller (MVC) architecture. It separates the application into three main logical components-model, view, controller. Each of the components are built to handle specific developmental aspects of the application. The model component corresponds to all the data related logic the user works with representing the data being transferred between the view and controller. View component is the user interface where it displays data to the user utilizing the model and also enables the user to modify the data. The controller component act as an interface between the model and view component to manipulate data using the model and interact with the view to render the final output. MVC provides a clean separation of concern, allows code to be tested easier, allows for better organization, scalability, extensibility, and code reuse.

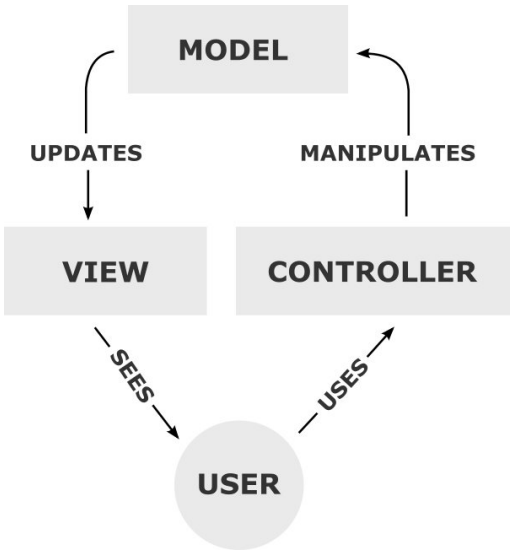

Figure 2: Model-View-Controller

2 Modular Decomposition

MODEL	VIEW	CONTROLLER
Board Move Player AI	Gameboard Sideboard	Mainstoryboard

Table 1: Modular Decomposition

BOARD:
This class is the model of the board. The Board.java class contains all the game data and game logic that is part of the game and user interaction. The board contains the game data of where all the pieces are on the board, the players on the board, the adjacent pieces as well all the possible milling combinations that are part of the Six Mens Morris game. This class was implemented in order to separate data from the user interface as well as the view controller.

MOVE:
This class is a model to simulate a move on the board. The Six Mens Morris has several different moves and having a separate model class for a move will allow for the board model to interpret the move and perform the necessary actions.

PLAYER:
The player model depicts a player object on the board. The player has four main properties that are useful in the game. The four main properties are color, numMills, numPiecesHolding and numPiecesOnTable.

MAINSTORYBOARD:
The MainStoryBoard is the controller for the game that connects the Board and the Views. The MainStoryBoard takes input from the GameBoardView (via getCurDestination()) and from SideBoardView (via getCurColor()) and applies a move to the Board game data. Therefore the MainStoryBoards main functionality it to take input from the user, intrepret it and modify the Board model accordingly.

GAMEBOARD:
The GameBoardView is the graphical user interface that displays information to user. The GameBoardView uses the Board data in order to determine what colours to fill each position.

SIDEBOARD:
The SideBoardView is a graphical user interface component that displays the available pieces to the user. The function of the SideBoardView allows the user

to select a piece to put onto the GameBoardView. The SideBoardView requires the game data from Board to determine how many pieces of each colour should appear to the user.

AI:
The AI class is an artificial intelligence player that can be implemented in the Six Mens Morris Game. The AI class uses a greedy algorithm where during each turn, the algorithm determines what is the best move based on a criteria set before hand. The AI algorithm has three methods that will return a move depending on the current phase.

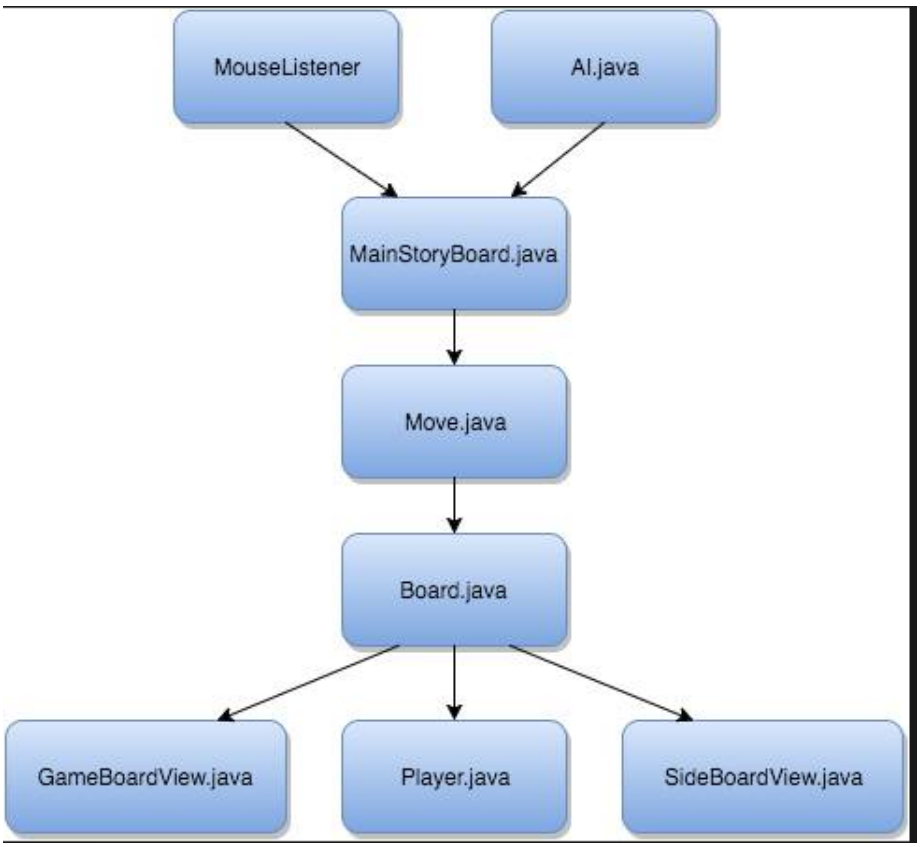

Figure 3: Module Hierarchy

3 Modular Guide

When decomposing the system into modules, the design document documents the module decomposition so that the developers and other readers can under-

stand the decomposition. This guide allows any reader to be able to understand how the program works with all of its interaction between modules without looking at the code.

3.1 MIS

The Module Interface Specification acts as a black box where the input is specified and an expected output is given. This does not go into detail of how the output is determined but instead allows readers who are not developers gain a basic understanding on how each module functions and interacts with other modules. Displays the public entities.

CLASS: Board
PACKAGE: Model
The Board.java class contains all the game data and game logic that is part of the game and user interaction. Contains access programs to update the state of the board and initialize the board.

INTERFACE

USES

Player Class Move Class AI Class

ACCESS PROGRAMS

getWinner(): Player
Returns the winner if there is a winner or null it is a tie or no winner.

getGameStateAtIndex(int i): int
Returns the gamestate at the desired index. Either empty, blue or red piece in the game state.

getCurrPhase(): int
Returns the current phase, either Move, Mill or Placement. Refer to Constants.java to determine which number represents each phase.

getPlayer1(): Player
Returns the data of Player 1.

getPlayer2(): Player
Returns the data of Player 2.

getCurrPlayer(): int
Returns the current player. Refer to Constants.java to determine which number represents each player.

getNumSpots(): int

Returns the number of spots available on the board.

getPlayer1NumMills(): int
Return the number of mills that Player 1 has.

getPlayer2NumMills(): int
Returns the number of mills that player 2 has.

getAdjacent(int piece): int[]

Returns an array of adjacent pieces (integer location) at a desired location.

setCurrPhase(int newPhase): void
Allows other classes to change the current phase of the board.

setPlayer1(Player player): void
Allows other classes to modify player 1.

setPlayer2(Player player): void
Allows other classes to modify player 2.

setCurrPlayer(Player player): void
Allows other classes to modify the current player.

loadGameData(String[] gameState): void
Allows other classes to load an external game data to the board.

Board(): Public
Public default constructor of the Board to create the Board object. This constructor creates a new empty game state, new player 1, new player2 and sets the default current player to player 1.

Board(int startingPlayer): Public
Public constructor of the board allows the option to indicate a starting player. Does exactly the same as the default constructor excluding the initial current player.

newGameState(): void
Sets all elements in the gamestate array to empty. Accomplishes this by a for loop from index 0 to gamestate.length-1 and sets the element to empty. Also this method creates a new Blue player 1 and a new Red player 2.

toString(): String
Converts the gamestate array into a line, each player into a line, the

currentPlayer and the current phase. Essentially saves all the important game data to a string.

CLASS: Move
PACKAGE: Model
This class is a model to simulate a move on the board. Contains access programs to get information about the mill and initialize the move.

INTERFACE

USES
None

ACCESS PROGRAMS
getFrom(): int
Returns the position a piece is originally at.

getType():int
Returns the type of the move

getPlayer(): Player
This method will return the player who made the move. This was implemented so that when the game implements the move it knows which player made it.

getDestination(): int
This method will return the destination of the move. This was implemented so the game can test to see if the desired position is empty and if it is place a piece in that position.

Move(): Public
This method will create a move for the selected player to the desired destination. This method was implemented so that moves can be made by each player.

CLASS: Player
PACKAGE: Model
This class is a model to depict the player's objects on the board. Contains access program to determine all of the properties pertaining to the player.

INTERFACE

USES
None

ACCESS PROGRAMS

Player(int color): Public
This method the constructor for the player class. It creates a player that is either blue or red. This was implemented so the game can have two distinct players (blue and red).

getColor(): int
This method will return the color of the player. This was implemented so that when moves are made on the board the game knows which color is supposed to be applied.

getNumMills(): int
This method will return the number of mills made by the player. This was implemented so the user can see how many mills they have made.

incNumMills(): void
This method will increment the number of mills made by one. This was implemented so there was a simple way to keep track of the number of mills made.

decNumMills(): void
This method will decrease the number of mills made by one. This was implemented in case there was a situation in which we needed to decrease the number of mills.

decreasePieceHolding(): void
This method will decrease the number of pieces in the players hand. This was implemented so every time the player places a piece on the board we can take one away for their hand.

getNumPieceHolding(): int
This method returns the number of pieces in the players hand. Each player starts with 6 pieces in their hand (9 in the case of 9 Mens Morris). This was implemented so we can keep track of how many pieces each player has in their hand.

getNumPiecesOnTable(): int
This method returns the number of pieces that are on the board.

CLASS: MainStoryBoard
PACKAGE: Controller
The MainStoryBoards main functionality it to take input from the user, intrepret it and modify the Board model accordingly. Contains access programs to communicate between the user input and the model.

INTERFACE

USES

Board Class GameBoardView Class SideBoardView Class

ACCESS PROGRAMS

MainStoryBoard(): Public
The constructor for the MainStoryBoard view creates a new game data model and creates calls createGUI() which creates the graphical user interface.

playGame(): void
This method starts the game. This method is made public so that the main server class can start the game.

CLASS: GameBoardView
PACKAGE: View
The GameBoardView is the graphical user interface that displays information to user. Contains access programs to determine which positions to show filled with a user's mill.

INTERFACE

USES

Board Class

ACCESS PROGRAMS

getWidth(): int
Returns the width of the board (standard usage for JComponent)

getHeight(): int
returns the height of the board (standard usage for JComponent)

getPieceX(int i): int
Each position has an associated X coordinate on the view, this will return the X coordinate of the desired piece at index i.

getPieceY(int i): int
Each position has an associated Y coordinate on the view, this will return the Y coordinate of the desired piece at index i.

setBoard(Board board): void
This public method allows for an outside class to set the board of the view. Mostly used for the view controller to update the view.

paintComponents(): void
Standard JComponent field that paints the graphical user interface.

resestDetination(): void
This public method allows outside classes, specifically the view controller, to reset the current destination that has been clicked.

getDestination(): int
When the user clicks on a position, that value is stored in a property of the board. This method will return the position the user clicked on.

mouseClicked(MouseEvent e): void
Standard method for a mouselistener, this method returns the closest piece the user has clicked on.

CLASS:Class: SideBoardView
PACKAGE: View
The SideBoardView is a graphical user interface component that displays the available pieces to the user. Contains access programs needed to display the mill on the side of the board.

INTERFACE

USES
Board Class

ACCESS PROGRAMS
getCurrentColor(): int
This method is how the view controller determines which colour the user has selected.

setBoard(Board gameData): void
This public method allows for an outside class to set the board of the view. Mostly used for the view controller to update the view.)

paintComponent(Graphics g): void
This is a standard method for a JComponent to paint graphics to the screen. In this method, blue and red pieces are shown to the screen.

getPreferredSize(): Dimension
This is a standard method for a JComponent that sets the size of the view.

SideBoardView(Board gameData): Public
This method is the constructor for the sideboard view. The sideboard takes in the gamedata, to have access to information about the game.

The constructor also takes in a parameter to denote the first player selected at random from the controller.

CLASS:Class: Constants
PACKAGE: Constants
This class is a constants resource class so that a common convention can be kept through the entire application.

INTERFACE

USES
None

ACCESS PROGRAMS
None

CLASS:Class: SixMensMorrisGame
PACKAGE: Default
This method represents the starting point of the program.

INTERFACE

USES
MainStoryBoard

ACCESS PROGRAMS
main(String[] args): Public
This method is where the program starts. This method creates the mainStoryBoard object then initiates the game play mode by calling playGame().

CLASS: AI
PACKAGE: Model
This class is a model to simulate the AI and its moves. Contains access programs to get information about the decision making process of the AI.

INTERFACE

USES
MainStoryBoard Class

ACCESS PROGRAMS
getPlayer(): int
Returns the AI player colour.

nextBestMillMove(Board board): Move
Returns the next best milling move. Searches through all the positions on the board that is the opponents colour and determines which piece to remove that is not in a mill.

nextBestMove(Board board): Move
TReturns the next best move during the movement phase. Uses calcMoveValue() and generateRandomMoveMove() to determine the next best move. If calcMoveValue() returns a -1, that is all the positions have equal strength, then a random move is generated.

nextBestPlaceMove(Board board): Move
Returns the next best placement move during the placement phase. Uses calcMoveValue() and generateRandomMove() to determine the next best placement move. If calcMoveValue() return a -1, that is all the positions have equal strength, then a random placement move is generated.

3.2 MID

The Module Implementation Design acts as a white box where the inputs and outputs are given with how they are determined. The MID relates to the the internal working of the module. These are usually looked at by developers and coders where they understand the level of abstraction and what it signifies. Both public and private entities are represented here. CLASS: Board

IMPLEMENTATION

USES
MainStoryBoard

VARIABLES
currPlayer(): Player
Each board has a current player that is chosen to make a move

winner: Player
Each board has a winner or a null player if the game is tied

currPhase: int
Each board has a current phase that can either by placement, movement or milling. The number is based on the number in the Constants class.

player1: Player
Each board has an associated player 1

player2: Player
Each board has an associated player 2

NUM_SPOTS: int
Each board has a legal number of positions on the board

gameState[]: int
Each position on the board has a game state, either empty or has a coloured piece in it.

adjacent[][]: int
A property of the board is that each position has associated adjacent position.

mills[][]: int
A property of the board is the limited amount of milling positions.

ACCESS PROGRAMS

getWinner(): Player
Returns the winner if there is a winner or null it is a tie or no winner.

getGameStateAtIndex(int i): int
Returns the gamestate at the desired index. Either empty, blue or red piece in the game state.

getCurrPhase(): int
Returns the current phase, either Move, Mill or Placement. Refer to Constants.java to determine which number represents each phase.

getPlayer1(): Player
Returns the data of Player 1.

getPlayer2(): Player
Returns the data of Player 2.

getCurrPlayer(): int
Returns the current player. Refer to Constants.java to determine which number represents each player.

getNumSpots(): int
Returns the number of spots available on the board.

getPlayer1NumMills(): int
Return the number of mills that Player 1 has.

getPlayer2NumMills(): int
Returns the number of mills that player 2 has.

getAdjacent(int piece): int[]

Returns an array of adjacent pieces (integer location) at a desired location.

setCurrPhase(int newPhase): void
Allows other classes to change the current phase of the board.

setPlayer1(Player player): void
Allows other classes to modify player 1.

setPlayer2(Player player): void
Allows other classes to modify player 2.

setCurrPlayer(Player player): void
Allows other classes to modify the current player.

loadGameData(String[] gameState): void
Allows other classes to load an external game data to the board.

Board(): Public
Public default constructor of the Board to create the Board object. This constructor creates a new empty game state, new player 1, new player2 and sets the default current player to player 1.

Board(int startingPlayer): Public
Public constructor of the board allows the option to indicate a starting player. Does exactly the same as the default constructor excluding the initial current player.

newGameState(): void
Sets all elements in the gamestate array to empty. Accomplishes this by a for loop from index 0 to gamestate.length-1 and sets the element to empty. Also this method creates a new Blue player 1 and a new Red player 2.

toString(): String
Converts the gamestate array into a line, each player into a line, the currentPlayer and the current phase. Essentially saves all the important game data to a string.

applyMove(Move move): void

		applyMove(Move move)
move.getType == Place	gameState[destination] == empty	Place piece, check for mill, switch player
	gameState[destination] != empty	Invalid move
move.getType == Move	isAnAdjacent[destination]	Move piece, check for mill, switch player,
	!isAnAdjacent[destination]	Invalid move
move.getType == Mill	isInMill[selected piece]&& !canDelete()	Mill selected piece
	!isInMill[selected piece]	
	isInMill[selected piece]	invalid move

canDelete(): boolean

For each index in the game state array do the following:

		canDelete()
gameState[i] == currentPlayer	!inMill[i]	True
i == gameState.length		False

countBlue(): int

For each index in the game state array, do the following:

	countBlue()
gameState[i] == Blue	returnVal += 1
gameState[i] != Blue	returnVal = returnVal

countRed(): int

For each index in the game state Array, do the following:

	countRed()
gameState[i] == Red	returnVal += 1
gameState[i] != Red	returnVal = returnVal

deletePieces(Player player, int destination): void

This method requires a color to delete and a destination that wishes to removed from the game board.

	deletePieces()
gameState[destination] == player.getColor()	gameState[destination] = Empty
gameState[destination] != player.getColor()	Invalid move, no deletion

canMove(): boolean

For all the elements in the gamestate array do the following:

		canMovet()
gameState[i] == currPlayer.getColor()	∃ adjacents[i] | adjacents[i][j] == Empty	True
i >= gameState.length		False

isAnAdjacent(int x, int y): boolean

	isAnAdjacent()
∃adjacents[] | x,y ∈adjacents[i]	True
else	False

movePieces(Move move): void

	movePieces()
gameState[destination] == Empty	Move piece
gameState[destination] != Empty	Invalid move, no move

isInMill(Player player, int position): boolean
For each index in the milling array, do the following:

		isInMill()
position ∈ mill[i]	∀ mill[i] == Player.getColor	True
	∀ mill[i] == Player.getColor	-
position !∈ mill[i]		False

alternateCurrPlayer(): void

	alternateCurrPlayer()
currPlayer == player1	currPlayer = player2
currPlayer != player1	currPlayer = player1

CLASS: Move

IMPLEMENTATION

USES

None

VARIABLES

type: int
Each move has an associated type, either milling, moving or placement

player: Player
Each move consists of a player associated with it. The player property determines which colour (player) made the move.

from: int
Each move has a move that a piece is originally at.

destination: int
Each move has an associated destination to move, place or delete a piece.

ACCESS PROGRAMS

getFrom(): int

Returns the position a piece is originally at.

getType():int
Returns the type of the move

getPlayer(): Player
This method will return the player who made the move. This was implemented so that when the game implements the move it knows which player made it.

getDestination(): int
This method will return the destination of the move. This was implemented so the game can test to see if the desired position is empty and if it is place a piece in that position.

Move(): Public
This method will create a move for the selected player to the desired destination. This method was implemented so that moves can be made by each player.

CLASS: Player

IMPLEMENTATION

USES
None

VARIABLES

color: int
Each player has an associated color property (either blue or red)

numMills: int
Each player has a property of how many mills they currently have on the board

numPiecesHolding: int
Each player can hold pieces during the first phase of the game. When both players are not holding anymore pieces, move to phase 2.

numPiecesOnTable: int
Each player can have pieces on the table. If the player has less than three pieces on the board, they are allowed to jump pieces. Use this property to determine when a user can jump pieces or if they have less than 3 pieces during phase two the game is over.

ACCESS PROGRAMS

Player(int color): Public
This method the constructor for the player class. It creates a player that is either blue or red. This was implemented so the game can have two distinct players (blue and red).

getColor(): int
This method will return the color of the player. This was implemented so that when moves are made on the board the game knows which color is supposed to be applied.

getNumMills(): int
This method will return the number of mills made by the player. This was implemented so the user can see how many mills they have made.

incNumMills(): void
This method will increment the number of mills made by one. This was implemented so there was a simple way to keep track of the number of mills made.

decNumMills(): void
This method will decrease the number of mills made by one. This was implemented in case there was a situation in which we needed to decrease the number of mills.

decreasePieceHolding(): void
This method will decrease the number of pieces in the players hand. This was implemented so every time the player places a piece on the board we can take one away for their hand.

getNumPieceHolding(): int
This method returns the number of pieces in the players hand. Each player starts with 6 pieces in their hand (9 in the case of 9 Mens Morris). This was implemented so we can keep track of how many pieces each player has in their hand.

getNumPiecesOnTable(): int
This method returns the number of pieces that are on the board.

CLASS: MainStoryBoard

IMPLEMENTATION

USES

Board Class GameBoardView Class SideBoardView Class

VARIABLES

gameModel: Board
This property is the game data model of the Six Mens Morris Game.

mainView: JFrame
This is the main JFrame view that holds the gameBoardView as well as the sideBoardView.

gameView: GameBoardView
This property is the view for the gameBoard.

sideView: SideBoardView
This property is the view for the sideboard.

infoLabel: JLabel
This label is a label to give information to user.

ACCESS PROGRAMS

MainStoryBoard(): Public
The constructor for the MainStoryBoard view creates a new game data model and creates calls createGUI() which creates the graphical user interface.

playGame(): void
This method starts the game. This method is made public so that the main server class can start the game.

createGUI(): void
This methods creates the GUI by putting the gameBoard view, sideBoard view, information label and verification button on the user interface.

saveGame(String filename): void
This method saves the game model into a specified filename. This method creates a file, and saves the toString() of the board data into that fall.

{filename == null && exists gameModel}
Procedure saveGame(String filename)
{filename != null}

saveGame(): void
This method saves the game model into a default filename. The method calls the saveGame(String filename) method and specifies a default filename.

loadGame(String filename): void
Given a specified filename, this method parses the data and creates a new board model. Essentially, this method does the opposite of saveGame() and sets the new board model to the gameModel.

{filename != null; gameModel = i}
Procedure (loadGame(String filename))
{gameModel = j}

loadGame(): void
This method loads the game model from a default filename. The method calls the loadGame(String filename) method and specifies a default filename to the function.

refresh(): void
This method refreshes the GUI to display the most recent graphical implementation of the data.

CLASS: GameBoardView

IMPLEMENTATION

USES
Board Class

VARIABLES

Height: int
Each view has a height property.

Width: int
Each view has a width property

outerSideLength: int
On a Six Mens morris game, there are two main rectangles, this property determines the length of the outside rectangle.

innerSideLength: int
On a Six Mens morris game, there are two main rectangles, this property determines the length of the inside rectangle.

gameBoard: Board
Each view has an associated data variable, in this gas the model for the view is the gameBoard.

curDestination: int

When a user clicks on the board, this variable stores the position that the user clicked on last.

pieceLocation[][]: int
Each piece location has coordinates on the view. This array stores all the x and y coordinates.

ACCESS PROGRAMS

getWidth(): int
Returns the width of the board (standard usage for JComponent)

getHeight(): int
returns the height of the board (standard usage for JComponent)

getPieceX(int i): int
Each position has an associated X coordinate on the view, this will return the X coordinate of the desired piece at index i.

getPieceY(int i): int
Each position has an associated Y coordinate on the view, this will return the Y coordinate of the desired piece at index i.

setBoard(Board board): void
This public method allows for an outside class to set the board of the view. Mostly used for the view controller to update the view.

paintComponents(): void
Standard JComponent field that paints the graphical user interface.

resestDetination(): void
This public method allows outside classes, specifically the view controller, to reset the current destination that has been clicked.

getDestination(): int
When the user clicks on a position, that value is stored in a property of the board. This method will return the position the user clicked on.

mouseClicked(MouseEvent e): void
Standard method for a mouselistener, this method returns the closest piece the user has clicked on.

fillPieceLocation(): void
This is a helper function to determine the x and y coordinates of each piece.

CLASS: SideBoardView

IMPLEMENTATION

USES

Board Class

VARIABLES

Height: int
Each view has a height property

Width: int
Each view has a width property

circlePositions[][]: int
This property stores all the locations of the pieces on the screen.

gameData: Board;
This property is the current game data for the view.

curColor: int
This property is the current colour the user has selected

ACCESS PROGRAMS

getCurrentColor(): int
This method is how the view controller determines which colour the user has selected.

setBoard(Board gameData): void
This public method allows for an outside class to set the board of the view. Mostly used for the view controller to update the view.)

paintComponent(Graphics g): void
This is a standard method for a JComponent to paint graphics to the screen. In this method, blue and red pieces are shown to the screen.

getPreferredSize(): Dimension
This is a standard method for a JComponent that sets the size of the view.

SideBoardView(Board gameData): Public
This method is the constructor for the sideboard view. The sideboard takes in the gamedata, to have access to information about the game. The constructor also takes in a parameter to denote the first player selected at random from the controller.

fillCirclePositions: void
This method populates the positions of each piece on the view.

CLASS: Constants

IMPLEMENTATION

USES
None

VARIABLES

Blue: int
Denotes the common convention for the colour Blue.

Red: int
Denotes the common convention for the colour Red.

Empty: int
Denotes the common convention for Empty.

numPeicesPerPlayer: int
Denotes the common convention for the number of pieces each player can have.

Place: int
Denotes the common convention for the phase Place.

Move: int
Denotes the common convention for the phase Move.

Mill: int
Denotes the common convention for the phase Mill.

ACCESS PROGRAMS

None

CLASS: SixMensMorrisGame

IMPLEMENTATION

USES
MainStoryBoard Class

VARIABLES

None

ACCESS PROGRAMS

main(String[] args): Public
This method is where the program starts. This method creates the mainStoryBoard object then initiates the game play mode by calling playGame().

CLASS: AI

IMPLEMENTATION

USES

MainStoryBoard Class

VARIABLES

player: player
Each AI has an associated player to denote the colour of the AI and the other properties that a regular player contains.

ACCESS PROGRAMS

getPlayer(): int
Returns the AI player colour.

nextBestMillMove(Board board): Move
Returns the next best milling move. Searches through all the positions on the board that is the opponents colour and determines which piece to remove that is not in a mill.

nextBestMove(Board board): Move
TReturns the next best move during the movement phase. Uses calcMoveValue() and generateRandomMoveMove() to determine the next best move. If calcMoveValue() returns a -1, that is all the positions have equal strength, then a random move is generated.

nextBestPlaceMove(Board board): Move
Returns the next best placement move during the placement phase. Uses calcMoveValue() and generateRandomMove() to determine the next best placement move. If calcMoveValue() return a -1, that is all the positions have equal strength, then a random placement move is generated.

newBoard(Board board): Board
Returns a new board form board so that modification of the board does not modify the actual board that the program is using.

getPlaceValue(int destination, Board board): int
Returns a value for a destination of the board. Assigns 4 points for forming a mill, 3 points for blocking a mill, 1 point for not placing beside an opponent, and -1 if none of the previous situations occurs.

generateRandomMove(Board board): int
Returns a random valid placement destination using board information.

generateRandomMoveMove(Board board): int[]
Returns a random valid movement using board information. Returns in an array where Array[0] is the from destination and Array[1] is the to destination.

calcMoveValue(Board board): int[]
Returns a valid movement in the form an integer array where Array[0] is the from position, Array[1] is the move score, and Array[2] is the destination.

4 Tracebility

The traceability chart will include a trace back of the requirements of the assignment in each of the class interface. This chart will allow a user to navigate the class interface to see how each class was implemented to ensure that the requirement was fulfilled correctly.

REQUIREMENT	ClASS LOCATION
Display a board of Six Mens Morris	Board.java, GameBoardView.java
Two sets of discs on the board	SideBoardView.java
Differentiation of the discs on the board	Board.java
Random order of play	MainStoryBoard.java (createGUI())
Start a new game	Board.java (newGameState())
Select colour of disk	SideBoardView.java (MouseListener)
Select position to place disk	GameBoardView.java (MouseLisener)
Move a disk onto position on board	Board.java(movePieces())
Ensure valid move	Board.java(applyMove())
Check if the player has won the game	Board.java (canMove() and applyMove())
Check if the game cannot be won	Board.java (canMove() and applyMove())
Display game state	MainStoryBoard.java (infoLabel)
Save Game	MainStoryBoard.java(saveGame())
Load Game	MainStoryBoard.java(loadGame())
Check if piece is in mill	Board.java(isInMill())
Have a menu to choose between player and AI	MainStoryBoard.java(createMenu())
Check for the best move for AI	AI.java(nextBestMillMove())

Table 2: Tracibility

5 Uses Relation

The uses relation diagram demonstrates the relationship between all the different modules in the program. This should be displayed in a hierarchy which shows clearly how one module relates to the others.

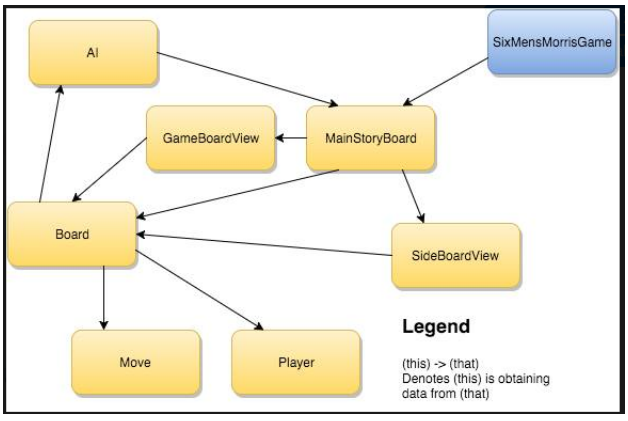

Figure 4: Uses Relationship

6 Anticipated Changes

6.1 AC #1: Changing Ruleset

One anticipate change that was looked at was the ability to change the type of game the user wants to play. The version the Six Men's Morris implemented in the assignment can be adjusted to have other versions, such as Nine Men's Morris or other variations. This would require the project to be able to adjust the view by allowing the user to have more or less pieces, allow for different win conditions, and allow for the board to be adjust to the amount of pieces available. By utilizing MVC, the project is able to have a separation of concerns that allows the developer to adjust the different variable to allow the user to enjoy different version.

6.2 AC #2: Colour Customization

Another small change that can be anticipated is the customization of the game. By separating what the user sees in the view, it allows the developer to allow the user to adjust the colour of the pieces or the colour of the board which does not impact the functioning of the game. This customization will allow the user further their enjoyment from the game. This possible change to the project may require also an additional menu to choose such customization.

6.3 AC #3: Resizing The Menu

The ability to resize the menu is another customization that is anticipated. This change will also achieve the goal to further the enjoyment of the user. Due to MVC, this change will not affect the controller or view but the module. Since the module stores the dimensions of the board and the objects on the board that define the view.

7 User Review/Evaluation

Evaluating the project and the design document that accompanies the project is an essential task that allows how the project is progressing. While testing asserts the functionality of the program, the review/evaluation should be a reflection of how well the design document delivers a comprehensible description of the entire project.

7.1 Understandability

7.1.1 What The Application Does and Purpose

The Six Mens Morris game is fully explained. The game's rules and pieces are given full explanations of how they work and what the user should expect. Any user reading this design document should be able to fully comprehend how to utilize this application and experience the game flawlessly.

7.1.2 Basic Functions of the Software

The interaction between the different classes utilized to allow this application to function is described in a comprehensible method. The document should give the user a clear picture of how each class follows the Model-View-Controller architecture and how the different classes are implemented, in terms of variables and functions.

7.2 Documentation

7.2.1 Quality

While the document completes each of the deliverables in a correct fashion, done in understable language to the common user, there is room for improvement. The document could reformat the explanations of the functions and variables of the classes in a more consise MID which is a universally recognized design tool. The modular decomposition could possibly be more effective in describing the relation between modules.

7.2.2 Completeness

All the required documentation set out by the requirements stated by the Assignment document has been fulfilled. Each response is defined for every stimulus. This includes everything specified in the deliverables and includes the testing document also.

7.2.3 Accuracy

The design document describes the design utilized to implement the application in a fully accurate way. All of the functions and classes in the modular decompositions have accurate descriptions associated with them and the uses relationship diagram displays the interaction of the various classes in our program in a factual basis.

7.2.4 Correctness

The design document clearly defines how each specification is fufilled and the locations of where the solution to the specification is found.

7.2.5 Clarity

One of the prime purposes in mind while doing the documentation was to convey the code clearly and explain it thoroughly. So each class, method, and function used was clearly and adequately documented to give the reader a clear guide to understanding the code completely. All the variables have been documented to facilitate easy follow through of the code as well.

7.3 Architecture

7.3.1 Extensibility

Our program has been designed with extensibility in mind throughout the different phases of the implementation. The program has the ability to have new functionality extended while not affecting the programs internal structure. We left room throughout the different classes for new modules to be added and the modularization of our design would enable additional methods throughout the application.

7.3.2 Modularity

The modularity of our design has been perhaps the most important element of implementation. We closely followed the model-view-controller design pattern and that helped us follow the requirements for the program. The MVC structure also allows for separation of concerns such that each section addresses a separate concern. The modularity that we used also aids in the extensibility of our program for possibly adding artificial intelligence.

8 Testing

The testing for our application was done throughout the implementation phase and we have encountered a huge variety of test cases through the different phases of development. The testing in this section is not function/method specific, but rather testing the requirements for the whole assignment.

	DESCRIPTION	EXPECTED OUTPUT	RESULT
Test 1	Run program by compiling on java	Runs without errors	Passed
Test 2	Run the program	Jframe window appears	Passed
Test 3	Check if JFrame size is 600 by 600	JFrame size is 600 by 600	Passed
Test 4	Check if player color selection is working	Red/Blue selections is correct	Passed
Test 5	Check buttons are working as expected	Buttons work as expected	Passed
Test 6	One piece is placed over another	Error message pops up	Passed
Test 7	Place piece on occupied node	No visual changes	Passed
Test 8	Load the game five times	Loads correctly without errors	Passed
Test 9	Place one color five times	Invalid game state	Passed
Test 10	Place a valid combination five times	Valid state displayed	Passed
Test 11	Place invalid combination five times	Invalid state displayed	Passed
Test 12	Reset the board five times	Resetting successfully performed	Passed
Test 13	Move a piece after all pieces have been inserted	Move allowed and performed	Passed
Test 14	Move a piece where another piece is located	Move unallowed, is not performed	Passed
Test 15	Form a mill with blue pieces	Blue wins the game	Passed
Test 16	Form a mill with red pieces	Red wins the game	Passed
Test 17	Move a piece not adjacent to former position	Move not performed	Passed
Test 18	Move a piece before all pieces have been placed	A new piece is inserted	Passed
Test 19	Run game until only 2 pieces left on board	Game won	Passed
Test 20	Save a game, after closing load that save	Saved game loads	Passed
Test 21	Place All the pieces P V P	Each player places 6 pieces	Passed
Test 22	Place pieces P V C	Each player places 6 pieces	Failed (fixed) AI had an extra move
Test 23	Run game until blue cant move	Red wins game	Passed
Test 24	Run game until red cant move	Blue wins game	Passed
Test 25	Run application for any visual bugs	Regular visuals	Failed(fixed) extra gameboard
Test 26	Form a blue mill	Remove one red piece	Passed
Test 27	Form a red mill	Remove one blue piece	Passed
Test 28	Save game with just jar file	Saves game	Passed
Test 29	Run game with computer	Computer only does valid moves	Passed
Test 30	Click 2 player	New game against player	Passed
Test 31	Click play with computer	New game against computer	Passed

Table 3: Testing

8.1 Tabular Testing

MainStoryBoard playGame() MainStoryBoard actionPerformed()

				Expected Output	Result
If (gameModel.GAMEOVER = True)	If (gameModel.getWinner().getColor() == Constants.BLUE)			Blue wins	Passed
	else()			Red wins	Passed
else if (this.gameModel.getCurrPhase() == Constants.PLACE)	If (PVC	Computer Turn)		Computer makes turn	Passed
	else if (sideView.getCurrentColor() != Constants.EMPTY && gameView.getDestination() != Constants.EMPTY)			Player Move	Passed
	If (Blue piece selected)			Place Blue Piece	Passed
	If (Red piece selected)			Place Red Piece	Passed
Else if (this.gameModel.getCurrPhase() == Constants.MOVE)	If(once)	If(Blue turn)		Select Blue piece	Passed
		Else()		Select Red piece	Passed
	If (PVC and computer turn)			Computer makes turn	Passed
		If (Blue turn)		Select Blue piece	Passed
		Else()		Select Red piece	Passed
	If (gameView.getDestination() != Constants.EMPTY)	If (currDestination != Constants.EMPTY)		Apply selected move	Passed
			If (Blue turn)	Select Blue piece	Passed
			Else()	Select Red piece	Passed
		Else () && if (gameModel.getCurrPlayer() == gameModel.getGamestateAtIndex (gameView.getDestination())	If (Blue turn)	Select Blue piece	Passed
			Else()	Select Red piece	Passed
		Else()		Error message please choose correct color	Passed
Else if(Mill is formed)				Message prompts to select opponent's colour	Passed
	If (Valid move selected)			Apply move for removal	Passed
	Else if (PVC and computers turn)			Computer removes a piece	Passed

Table 4: Tabuler Testing

			Expected Output	Result
If (New game clicked)			Start new game	Passed
Else If (Load game clicked)			Enter file name	Passed
	If (file name ==)		Unable to locate file message	Passed
	Else ()	If (file is found)	Load game	Passed
		If (File is not found)	Unable to locate file message	Passed
Else If (Save game clicked)			Enter file name	Passed
	If (File name ==)		??	??
	Else ()		Save game	Passed
If (Exit is clicked)			Exit game	Passed
If (Play with computer is clicked)			New game against computer	Passed
If (2 player is clicked)			New game against player	Passed

Table 5: Tabuler Testing

9 Appendix A

List of Figures

10 Appendix B

List of Tables

11 Appendix C

What Has Changed
Added the AI class to the module decomposition, MIS, and MID. All graphics have been updated to include this class. Uses relationship that pertain to the AI class in the rest of the program have also been updated. MID of MainStoryBoard had pre/post condition statements added to some of the methods. The tracability chart has been updated to include new requriements. Anticipated changes have also been updated. Testing for additional requriements and adding tabular testing.