Phillip Pavlich 001414960
Prakhar Jalan 001450321
Dinesh Balakrishnan 001409123

2AA4 ASSIGNMENT 2 DOCUMENTATION

NOTE: New changes to the documentation is indicated with a (***)

Program Assumptions: Only able to save 1 previous game at a time

5.1. a description of the classes/modules you have decided to use in your application, and your explanation of why you have decomposed the application into those classes;

Description of classes/modules:

There are four classes used in our application.

- moveLocations (1)
- PlayerPiece (2)
- SixMensMorris (3)
- StdDraw (4)
- gamePlay (5) ***
- SaveGame (6) ***

(1) This object class is used to be able to map out the positions on the board to the pieces and then determine whether these spots are occupied or not. The primary characteristics of the moveLocations object are the " box coordinates" of the clickable location (minX, maxX, minY, maxY) and its occupancy.

(2) This object class is used in order for each piece to have its own individual characteristics initially defined and modified throughout the game. By having a color and a piece number, we can accurately identify the specific piece of each player, as well as modify its position as the game goes along.

(3) This main class is responsible for creating the user interface (calling methods from StdDraw), creating the 16 moveLocation objects, the 12 PlayerPiece objects, and combining everything into a runnable program. The main method calls a variety of branching methods such as newGame()

in order to maintain an organized set of code. ***The continue game method has also been implemented, as it also utilizes the gamePlay class's methods to continue a game (after pieces have been placed for continual).

(4) This module is used in order to gain access to drawing tools for the game's user interface. Several methods within this library were used to design the main menu screen as well as the game board and its pieces.

(5) ***This new class takes care of the entire new game function; it allows the players to face one another by placing and moving pieces, milling their pieces, and finally ending the game appropriately. This class specifically targets the aspects of piece movement, milling, and game results. The placement of the pieces is taken care of in the main SixMensMorris class. Through creating a position matrix, a variety of case statements, and manipulating the graphics accordingly, this class carries out the said tasks very efficiently.

(6) *** This new class allows the user to save a particular game (after piece placement) at the end of any turn. Once the button is clicked, a file is created in order to later be read for re-initializing a game given specific details; this includes which player's turn it is, the number of pieces, and where each piece is located on the position matrix.

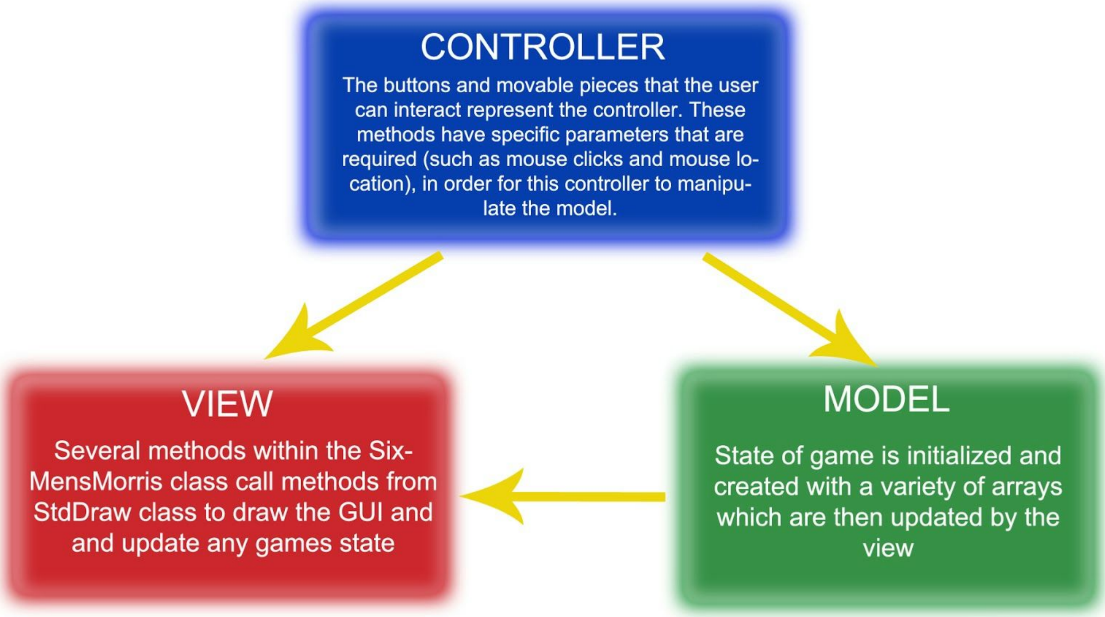
Decomposition Explanation:

By decomposing the application into these 3 classes (StdDraw given module), we organize the program in a clear and understandable manner. Specifically, the two object classes are made to retrieve individual data regarding the 16 possible movable positions and the 12 pieces used within the game. This makes it easy to identify and modify the state of the game by manipulating these objects accordingly. Additionally, it is evident that the main class (SixMensMorris) is used to organize the information needed in the game and sequentially develop the game's state. For example, the drawing of the board is done separate from the button functions themselves (newGame). The continueGame function is structured quite differently than the newGame, as it allows the user to place as many pieces as they require, and informs them of any errors. *** Furthermore, both these methods utilize methods from the gamePlay class in order to carry out the game accordingly. The key difference is that newGame systematically (turn based) allows players to place pieces, while the continueGame allows players to cooperatively recall their previous game. Both are followed by piece placement, mills, and game results. *** By separating the classes in this specific way, we easily create all the objects needed into the newGame method and continueGame method, store them in their own arrays for later use, and then read the user's mouse clicks corresponding to the locations they wish to place their pieces.

*** Additionally, the gamePlay class allows players to actually move their pieces and essentially applies the "rules" of the game and corresponding results along the way, thus justifying its separation from the other classes. The createGame class itself efficiently carries out piece movement and keeps track of several object attributes (moveLocations and PlayerPiece) as well as modifications made thereof. There are certainly other ways of approaching this application, but we chose to go with our intuition and attack the problem with our initial concept of the class layout.

The buttons and movable pieces that the user can interact represent the controller. These methods have specific parameters that are required (such as mouse clicks and mouse location), in order for this controller to manipulate the model.

Figure 1: The MVC Architecture Diagram


CONTROLLER
The buttons and movable pieces that the user can interact represent the controller. These methods have specific parameters that are required (such as mouse clicks and mouse location), in order for this controller to manipulate the model.
VIEW
Several methods within the SixMensMorris class call methods from StdDraw class to draw the GUI and and update any games state
MODEL
State of game is initialized and created with a variety of arrays which are then updated by the view

5.2. for each class, a description of the interface (public entities), and make sure that there is a description of the semantics (behaviour) of each public method in the class including a reference to what requirement(s) was(were) implemented in the method, as well as a description of the syntax;

- moveLocations (1)
    - Constructor (1.1)
    - getMethods (1.2)
    - checkOccupied and changeOccupied (1.3)
- PlayerPiece (2)
    - Constructor (2.1)
    - getMethods (2.2)
    - changePiecePosition (2.3)
    - changeEnabled (2.4) ***
- SixMensMorris (3)
    - Main method (3.1)
    - mouseInRange (3.2)
    - mainMenu(3.3)
    - setBoard (3.4)
    - newGame (3.5)
    - continueGame (3.6) ***
- gamePlay (4) ***
    - resetMill (4.1)
    - turnButton (4.2)
    - millBoard (4.3)
    - createGame (4.4)
- SaveGame (5) ***
    - saveButton (5.1)
    - saveInfo (5.2)
    - buildGame (5.3)
- StdDraw ~ Given module

(1.1) This constructor initializes and creates the moveLocation object based on the parameters given: minX maxX minY maxY. The occupancy is also initialized to be false when the object is created.
(1.2) The 4 getMethods are the accessor methods for each bound of the clickable location (minX maxX minY maxY), where the object's corresponding characteristic value is returned. *** The new getMethods called getMidX and getMidY were also created in order to calculate the middles of the clickable movelocations.
(1.3) These occupancy methods are for returning boolean values (true or false) regarding whether the spot itself is occupied or not (checkOccupied) and modifying its occupancy (changeOccupied).
(2.1) This constructor initializes and creates the PlayerPiece object based on the parameters given: color and piece number. Also, the piece's activation is initially set to false and the piece's position is set to 0 (not placed).
(2.2) The 3 getMethods are the accessor methods for each of the primary characteristics of the PlayerPiece object. Its color, piece number, and piece position are returned when these getMethods are called (respectively).
(2.3) This method takes in a new position as a parameter, and modifies the piece's current position to this new value. This allows the piece to essentially "move" upon the board as its position is needed as a characteristic (backend).
***(2.4) We added this method in order to deal with pieces that were "removed/milled" from the board, thus they would become disabled. We simply modify the private object variable that is of type boolean (enabled).
***(3.1) Since this is the main method, everything here is directly related to the user interface and the basic sequential structure of the program. We start of by calling the mainMenu method to build a mainMenu screen using StdDraw. Next, we check whether the use is clicking on a button or not, and if so, take them to that corresponding screen. The newGame method allows players to place their pieces upon the board accordingly, while the continueGame method allows them to set up their previous board, and inform them of any setup errors thereafter. Players can then play the game, and/or save their game at the end of any turn.

(3.2) This method returns true only if the mouse is within clickable bounds of a given location. The method takes in the mouse location (x,y) and compares it to the given (minX maxX minY maxY) values to determine whether the mouse is in range.
(3.3) This uses StdDraw to draw the entire main menu. The methods are used directly from StdDraw to create the two appropriate button pictures, and other visuals. This method is purely for visuals.
(3.4) This uses StdDraw to draw the game board. The methods are used directly from StdDraw to create the two appropriate board, player pieces and other visuals. This method is purely for visuals.
(3.5) This method starts off with one of the player's going first (randomizer), and then alternates their turns to place their pieces upon the board. Here, the PlayerPiece objects and moveLocation objects are created, and then every case of clicking (position placement) is checked; once the user places their piece, the program modifies specific values to shift the state of the game appropriately.
***(3.6) This method starts off with the user being able to place blue or red pieces upon the board (to their liking), and inform them of any possible errors (once they click the done button). To swap colors, we created two buttons (blue and red), so that they can switch at anytime. In the method itself, the placement occurs within a while loop that breaks once the user clicks the done button, and then displays text concerning any possible errors. The user is then able to continue their game thereafter through placing their pieces turn by turn, milling, and producing an end game. At any turn during the game (after piece setup), players are allowed to save their current game and later reopen it when they wish to.
*** (4.1) Saves the information of the mill if one occurs. It also checks if a previous mill has been changed so that it is valid once again.
*** (4.2) This method creates the mill matrix, and checks if there are any mills on the given board. If so, then it allows the user to mill one of the opponent's pieces.
*** (4.3) This button allows the player (in setup for the new game) to proceed to the next turn of the game.
*** (4.4) This allows users to move pieces, mill pieces, and produce an overall end game (result). It runs through by alternating turns.

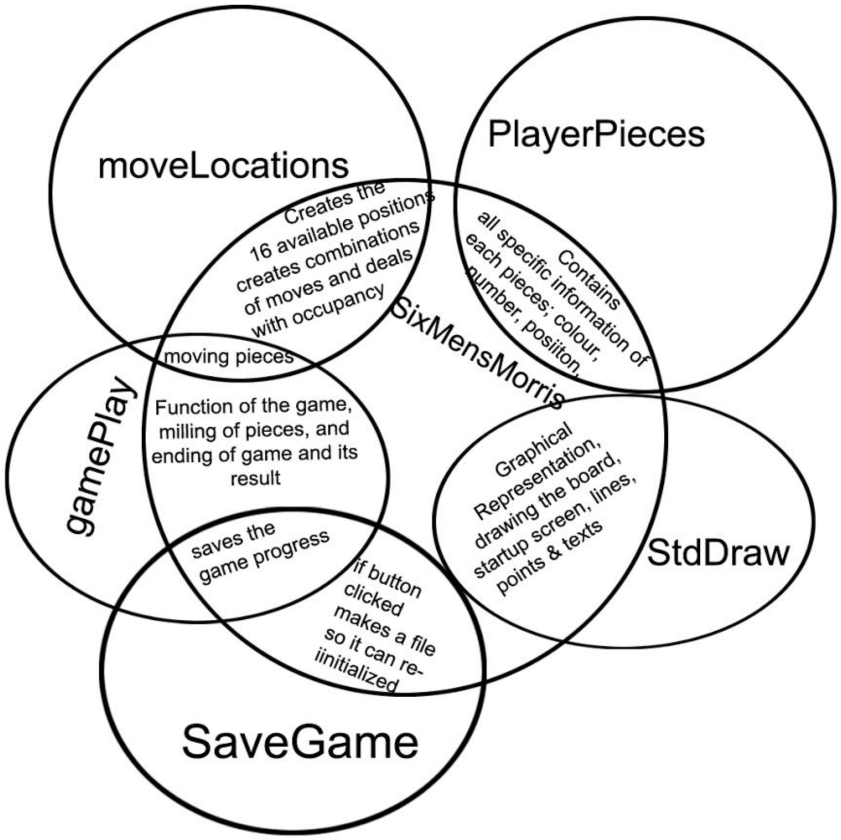
*** (5.1) This is a method that is displayed throughout the movement phase of the game, and allows the user to save their game at the end of their turn if it clicked.
*** (5.2) This method allows all the data to actually be saved; which includes whose turn it is, the number of pieces for each player, and their appropriate positions. It saves this information to a file called "savedgame.txt".
*** (5.3) This method essentially builds the game from the file (reads the file), and thus re-initializes the overall game state.

5.3. a view of the uses relationship;

There are many uses of relationship in this application. Everything relates back to our SixMensMorris class. A simple relationship can be seen by the interaction of the startup screen where there is the option to go to "Start New Game" or "Continue New Game". If new game is clicked, it will start a new game from scratch while they can start a game where they can play all the pieces beforehand to start a game and they will also have an option to check if there is any error, if they decide to click on continue game. SixMensMorris uses Std draw to set up the graphical representation of the screen and the game board. It uses the StdDraw class to draw out the startup screen, the game board with all the lines and points, the pieces and even the text. The moveLocation class relates back to SixMensMorris as it first creates the 16 positions available for the pieces to be clicked on by the mouse. Then, it will also make sure only one piece will go and occupy the position and not let the rest of the colours go and will stop from future pieces to get that spot as it will be occupied. This class also lastly, make sure there will be less available spots and the combination of available spots will be o3ne less for the next team. Another relationship will be one between the PlayerPiece class and the SixMensMorris. Here, they interact as they take the player pieces from this class as the playerpiece class contains specific information of each piece such as the colour, number, and position. The SixMensMorris class then will take this information and collect all 12 pieces. It also chooses who will go first and who will go second.

***An important relationship and strongest one is the one between the gamePlay class and the SixMensMorris class. This class is where we see the main correspondence as we see the the game function in this class throughout this class. The players are able to take turns after each other. This also correlates with the moveLocation about the movement of pieces. This relationship will also take care of milling and the gameplay.It will also then deal with the ending of the game and game results. A final relationship is with the final class called SaveGame and the SixMensMorris, this relationship deals with saving the progress of the game. This is a big relationship with the gamePlay class which is responsible for the primary game mechanics. With a click of a button, a file will be created and later read in order to re-initialize the overall game state.

Figure 2: This Figure Shows the Uses of the the Relationship in a Venn Diagram


moveLocations
PlayerPieces
Creates the 16 available positions creates combinations of moves and deals with occupancy
Contains all specific information of each pieces: colour, number, positon.
SixMensMorris
moving pieces
gamePlay
Function of the game, milling of pieces, and ending of game and its result
Graphical Representation, drawing the board, startup screen, lines, points & texts
StdDraw
saves the game progress
if button clicked makes a file so it can re-iinitialized
SaveGame

5.4. for each class, a description of the implementation (private entities), including class variables - include enough detail to show how the class variables are maintained by the methods in the class;

For the moveLocations class, there are 5 private entities. There is the double minX, double MaxX, double minY, double maxY and lastly the boolean occupied. These private entities are maintained through the accessor methods within the class through getMethods. The first 4 methods return the current object's corresponding values, while the 5th method (occupied) maintains the object's position occupancy upon the board.

For the PlayerPiece class, the private entities are the String pieceColor, int pieceNum, boolean pieceActivated, int pieceRadius, and the double piecePosition. These 5 entities are implemented and maintained through the accessor method within the class with getMethods. The pieceColor, pieceNumber, and pieceRadius are private entities that will remain unchanged, as we only need to access this information to identify individual PlayerPiece objects. The piecePosition and pieceActivated however will change as the game progresses, since these need to be modified based on the user's decisions of piece movement/placement. They are easily maintained through the use of the getMethods, as the piecePosition can be modified when given a new position value, and the pieceActivation is initiallized as false due to being prior to placement.

5.5. an internal review/evaluation of your design.

We started the design process by brainstorming how to implement an interactive game like Six Men's Morris. We came to an agreement that StdDraw would provide a nice colourful design that looks appealing in which we can still implement an interface for the user. We wanted to modularise the code so that it is broken up into tiny bits of code. When these modules are put together, it creates our game. Creating classes for pieces and locations on the game board were a crucial step in our game design. These allowed us to keep track of all the pieces as they are moving around the board. We are also able to check if a location is occupied in order to prevent a player from putting a piece on top of another piece.

Get methods for the pieces and locations came in handy for allowing the user to click on locations or pieces that they wish to place or use. We kept some variables inside our classes private to implement separation of concerns. We knew that it was important to keep parts of the code unknown to the user to protect our implementation. I think our implementation of our design went a long way to making assignment 2 and 3 easier. We removed potential problems and made our code have a strong structure to it.

Our design also consists of a continued game where the user can place as many pieces for both sides as he wants in locations. If the user sets up the board in a way that is invalid, error messages are displayed. The two possible errors are if the user placed more than 6 pieces for one side and if less than 3 pieces on one side were placed. If less than 3 on one side, it is a sign that the game is over or that the players have not finished placing their men on the board.

*** Furthermore, the new gamePlay class fits the design well in regards to separation of concerns. It efficiently carries out the tasks player piece movement, mills, and game results. The specific design of the primary method developed in the class (createGame) allows us to easily see where and what function is addressed. The matrices used in this class (position matrix and mill matrix) further illustrate that the class design is sufficiently understandable, and easily mentally graspable (overall layout). A variety of logic errors are also appropriately addressed; the user interface itself can prove to raise several issues regarding how the players wish to interact/move their pieces. Several cases were covered, including mouse clicks upon illegal pieces, and legal moves to be made, as well as milling pieces off the board accordingly. Regarding the physical aspects of the design, we implemented a “Next Turn” button in order for both players to alternate their turns efficiently as it indicates a break in the turn based game, therefore creating a well structured game in terms of functionality and overall game mechanics.

5.6. Document the code so that it is clear how the code follows its design, and also explain design decisions in the code that were not included in the design document.

To document the code, we have put some comments into our code for each class and method. Before each method is a description of what the method does. We have explained all the design process and decisions on how to implement this game in the previous sections, especially 5.5.

6. Include a test report document that records how you tested your application (we have not discussed testing yet – so you are on your own with this document).

We tested our design and code using numerous strategies and criteria. First off, when the code is run there is no syntax errors meaning that we have a functional code. We designed our code with StdDraw so that we can provide the user with an attractive program that also uses interactive features. StdDraw does not have buttons so we used methods from that class to make our own buttons. We made it check if the mouse was clicked and the x and y component were both situated inside the button. To test this, we ran the code and tried to click the button and it continued us to the next step in the game. When we did not click the button it keeps going until you do just like it is supposed to.

We tried to play the game, in order to check if the player's turns alternate as they set up the board with their pieces before they start moving the pieces. We also did a test to click the start new game button to make sure the board is setup correctly on the window. The board needed to have straight lines connected for the two squares and little dots and all possible locations that you can put a piece. One aspect of this game that is not legal is to place a piece on a location that is already occupied by another piece. To test this, we tried the setup of our game and clicked on an already occupied location. The game will not allow you to make a move there as it is invalid. We tested all the get methods and mutators in our game pieces class as well as our board location class. We printed out the values after calling these methods to ensure we got the expected output.

We tested the randomizing of which player goes first. We played several games to ensure that there were some games that Blue was able to go first and others where Red was able to go first. We also tested our ability to overwrite already drawn elements on the window. To get rid of an element, we made the assumption that you can draw something over top of it to hide the element. Our hypothesis was correct as we were able to draw rectangles over top of unwanted elements to make that element disappear. Our continue game button is a button that allows the user to put pieces where he would like to set up them up and then click done. Once he clicks done, it will output any errors in the board setup. The possible errors are that one player doesn't have 3 players on the board meaning the game is won or the setup hasn't finished. The other error is that more than 6 pieces were placed on one side. We have tested the errors to make sure they are printed out for those two cases. Another factor that we considered when testing was the positioning of the elements. It is important that all circles are centered on the location so that it doesn't mess up the implementation of the game. To test this, we have clicked on all possible piece locations to ensure that every piece that is placed on the game board lands on the proper location.

Testing is one of the most important parts of coding in order to ensure correctness and efficiency. We have made sure to do all the necessary tests to prove that our code creates a user friendly interactive setup to the game Six Men's Morris and allow users to place their pieces on the game board.

***For the second assignment, we added in the option to play the whole game. From the continued game portion and from the new game portion, we are able to play the game to the end. We had to test all the possible endings. We tested if red wins because blue has less than 3 pieces. We tested if blue wins because red has less than 3 pieces. We also tested if one player is cornered and unable to move which means the game is over and that player loses. We also had to test if mills(3 pieces of the same color in a line) were functional. We tested the mills through continue game and new game and even with multiple mills at a time. We also prevented a player who earns a mill from removing a piece from the other team that is in a mill as well unless they have no other pieces. This was a rule when we looked up the rules to Six Men's Morris. Finally, we checked if each piece was able to move to an adjacent, vacant spot when it is their turn. This test case was also passed.

***Another key feature that we had to perform tests on was the SaveGame class. This is a new class that allowed the user to click a button and save data about the current game so that they can play later. We ran tests to make sure that when the user clicked the button, all data was saved to a text file named ‘savegame.txt’. Furthermore, we tested the load game button on the main screen to see if we could reload the data that we saved in order to play from where we left off. In testing, we found out that loading the game brings it back to the same screen and you can continue playing with mills still functional until one player wins the game. We passed all our test cases to show that our code is functional and meets the requirement specifications.