Assignment Sprint 2 - Rationale for Complete Class Diagram

Key Classes

The first “key” class our group chose was the *GameManager* class. There are many different explanations for this. Based on the MVC architecture model, this is a controller class that communicates between model and view classes. This class is accountable for processing the click action that the user does on the graphical user interface (GUI) and invoking necessary actions based on the click. Additionally, this class is charged with invoking methods that could display the action on the GUI. Because all of the action takes place on the *GameBoard*, and the *Display* class is used to display the user what's going on. Consequently, *GameManager* has a relation with the *GameBoard* class and the *Display* class.

The line of code that you just looked at belongs to a method in the *GameManager* class. This method accepts an *Event* as an argument and then processes that *Event*. Then, depending on the occurrence of the event, it performs the action denoted by the code, such as the Slide action or the Place action. Lastly, we will use the *Display* class to refresh the graphical user interface.

It is possible to view this class as an intermediary between the backend and the frontend of the website. Because there are multiple actions that can be carried out on the gameBoard, such as remove, move, jump, etc., and that each action will cause result in different display on the GUI, it was not possible to make it a method because doing so would have caused too much work on a method, which would have been difficult to maintain and debug. In addition, there would be a violation of the "Separation of Concern" notion due to the fact that a method is needed to carry out a single operation. The creation of a class would allow us to divide up larger functions into several smaller ones that are each dedicated to a different set of responsibilities.

The second “key” class our team chose was the *Canvas* class. This class is responsible for drawing out all the GUI components on the main window, and will help the *Display* class update all GUI elements whenever the state of the game changes.

The functionality of GUI drawing was isolated from the *Display* class into its own class - Canvas, as if it were a single method, the *Display* class would violate the Single Responsibility Principle. *Display* class is already in charge of initialising the main window, and catching mouse events that occur on that window, so that it can pass information of the player behaviour to the *GameManager* class. If it also had to handle painting and repainting GUI elements on top of that, it would become a God Class since it handles more than two responsibilities, and the code would become too convoluted.

Key Relationships

Game and GameManager

The *GameManager* class is responsible for handling the actions the players perform, and changing the player's turn. *GameManager* communicates with the *GameBoard* class to update the pieces on the positions of the game board, and the *Display* class to update the GUI view of the 9 Men's Morris game. The *Game* class is responsible for starting the 9 Man's Morris game and ending the game when the win conditions are met, i.e. when the opposing player has less than 3 pieces left on board or no legal moves can be made.

The *Game* class has a one-to-one association with the *GameManager*, because the *Game* class will initialise the main game loop and game details when starting the game, before passing control to the *GameManager* to manage and update the state of the game.

Moreover, the *GameManager* needs a strong relationship with the *Game*, as it needs to know the *Players* initialised by the *Game* class to correctly increment or decrement the number of pieces each player has on hand and on the game board.

Display and Canvas

The *Display* class is responsible for initialising the window where all the GUI elements of the 9 Men's Morris game are being displayed, and handling all the GUI elements. The *Canvas* class is responsible for drawing the GUI elements on the window initialised by the *Display*.

The *Display* class has a one-to-one association with the *Canvas* class. This is because the *Display* class needs to call the methods in the *Canvas* class to draw the elements. For example, when the player is performing the action to place a piece onto a position on the board, the *GameManager* will verify if the action is valid. If the action is valid, it will call a method in the *GameBoard* to add the piece onto the position the player selected and tell the *Display* to update the GUI view of the game. The Display will request the *Canvas* to draw the piece onto the game board. The *Display* class is a composition of the *Canvas* class because the *Canvas* class cannot functionally exist without the Display class.

Inheritance

Inheritance is useful for allowing new child classes to reuse code and variables from existing parent classes, in order to reduce code duplication and support polymorphism. It is typically used when the parent and child classes share some common attributes and behaviours. There are three instances of inheritance that were used in the design of our 9 Man Morris Game, which are the classes in the actions, players, and model packages.

For the model package, the classes *RealGame* and *TutorialGame* extend the *Game* Abstract Class. The two types of games share similar variables and methods (such as both having *GameManagers* and *Players*), but they are variants of the 9MM Game that have their own functionalities (Eg. Tutorial Game operates slightly differently than a RealGame, as it teaches a new player how the rules of the game works).

Likewise for the players package, *HumanPlayer* and *ComputerPlayer* extends the abstract *Player* class, as variants of a typical 9 Man Morris Player. Both types are valid players in the game, but *HumanPlayers* would play the game by mouse input via the GUI (See view package - *Display* and *Canvas* Classes). Meanwhile, *ComputerPlayer* would be a bot that automatically makes moves on its own based on the state of the game.

The third use of inheritance is for the various action classes for *PlaceAction* (Placing a Piece), *RemoveAction* (Removing a Piece), *JumpAction* (Jumping A Piece during a Mill), and *SlideAction* (Sliding A Piece from one position to another adjacent one). All the action classes extend *PieceAction* as valid actions that can be performed on pieces on the gameboard.

Inheritance in our design supports polymorphism as it allows the *Game*, *Player*, or *PieceAction* to have many different variants, making the code reusable and dynamic. This allows the code to easily adapt to new requirements or changes, as whenever a new variant of *Game*, *Player*, or *PieceAction* Class is created, the new variant can just extend its respective parent class for easier implementation, instead of being built from scratch.

Two Sets Of Cardinalities:

GameManager and Player

Each instance of the *GameManager* class can be connected with three distinct instances of the player abstract class, according to the cardinality 1-3 that exists between the *GameManager* class and the player abstract class. In the case when the *GameManager* has the attributes player1 and player2 in addition to the currentPlayer property, the currentPlayer attribute's job is to keep track of which player's turn it is in the game at any given time. A snip of the code can be seen below.

The attributes of player1 and player2 are used to represent the two players in the game. By evaluating the player1 and player2 attributes, the currentPlayer property determines which player's turn it is to perform a move.
By keeping track of which player is the current player, the game manager can ensure that each player has an equal opportunity to make a move. This allows the game management to alternate between the turns of the two players, and to be played fairly and also makes the experience a pleasant one for those who participate in the game.

Position and Other Positions (Neighbours)

Given that the cardinality between the *Position* class and itself runs from 1 to 4, each instance of the *Position* class can be associated with 1 to 4 more instances of the *Position* class. This means that a position on the game board may have 1 to 4 adjacent positions, depending on where it is located on the board. A location at the corner of the board, for example, will have just two neighbouring positions, but a position in the centre of the board will have four adjacent spots.

As the game maintains track of the positions on the board that are adjacent to each position on the board, it also indicates the player's next moves when they click the hint button. Moving on, because it would be difficult to perceive all of the possible moves on the board at the same time essentially when both players have in total more than three pieces on the board.

Design Pattern: Chain of Responsibility.

The design pattern that we employed in our 9 Man Morris Design is the Chain of Responsibility Pattern. In our chain of responsibility, the inputs of the HumanPlayer (Client) through the GUI (painted by Canvas), are represented as Mouse Events (Mouse clicks, etc.), which are captured by the Display Class. Display Class then sends the data to the GameManager Class. The GameManager identifies the Action that the player is performing from the mouse data, and invokes the relevant Action Classes / Gameboard Classes to manage game states / execute instructions. This enables control over the order that mouse behaviour from the clients are handled.

<u>Two other feasible alternatives</u>

One alternative to implementing this functionality is using the State Design Pattern where there will be a state interface that defines how the game behaves at different states. The *GameManager* will handle the actions performed by the players differently based on the current state of the game. For example, at the start of the game, the state of the game will be placing pieces. The *GameManager* will only call methods that are related to placing the pieces onto the board. However, for this design pattern, we would need to define each state of the game clearly and carefully. Otherwise, it can make our code harder to maintain and increase the complexity. Additionally, by defining various actions as classes with each *Action* class knowing it's respective action, *GameManager* just needs to invoke those actions to perform their respective functions. This will reduce the amount of code and responsibilities in *GameManager*, enforcing separation of concerns better. If the state alone is defined as an Enum, then additional code for the logic of managing the states would crowd the *GameManager* class.

Another alternative is to use the Observer Design Pattern. The *GameManager* notifies the *GameBoard* and *Display* of game state changes. The *GameManager* will notify the *GameBoard* to add the player's piece to the game board and the *Display* to draw the piece on the GUI view when the player places a piece. This design pattern may not work for the game because the *GameBoard* and *Display* will be notified randomly. The *Display* will call a *Canvas* method to draw all the pieces the players have placed based on the *GameBoard's* list of placements, which will store a piece if there is one. This can cause issues because the *Canvas* won't draw the piece on the board if the *GameBoard's* list of positions isn't updated first by adding the piece to the player's position.