DOMAIN MODEL RATIONALE

Player

Reason for choosing this domain
Since both HumanPlayer and ComputerPlayer will later have the same attributes (such as an arrayList of tokens) and methods (such as the basic moves of moving the token), Player is introduced as a parent class for both HumanPlayer and ComputerPlayer. It defines the standard for all the actions any type of player can take during the game, hence, Player will be defined as an abstract class so that there is no repetition of code in the HumanPlayer and ComputerPlayer classes. Having an abstract super class for Player, ensures that each type of player can have its own implementation details within the same domain.

Discarded Alternative: The HumanPlayer and ComputerPlayer classes could repeat the same skeleton code for a Player's base actions but that would violate the DRY principle. Hence, an abstract class was a better option as most of the elements can be reused and some method implementations can be made mandatory (by using abstract methods). This solution also applied the Open-Closed Principle since extension of basic Player behaviours is easy and such changes do not need to be copied to multiple classes.

Reason for the relationships

- Player – Move : Player can use Move to move their tokens at each turn to any valid position.
- Player – Token : Each Player has a total of 9 tokens at the start of the game. It might be reduced slowly one by one to a minimum of 2 tokens in which the game will end.

HumanPlayer

Reason for choosing this domain
The 9 Men's Morris game requires at least one HumanPlayer for it to be runnable. The HumanPlayer domain defines the implementation of any behaviours that any user of the game does

Reason for the relationships

- HumanPlayer – Player : HumanPlayer is chosen to be a child class to the Player in order to prevent repetition of items in multiple player classes. It extends from the abstract Player class.
- HumanPlayer – Game : A Game will have at least one HumanPlayer (ComputerPlayer Mode) and a maximum of 2 (HumanPlayer Vs HumanPlayer) depending on the game mode chosen.

ComputerPlayer (Advanced Feature)

Reason for choosing this domain
The user can select to play against a bot whose actions and behavioural implementation is defined by the ComputerPlayer domain.

Reason for the relationships

- ComputerPlayer – Player : ComputerPlayer is chosen to be a child class to the Player in order to prevent repetition of items in multiple player classes. It extends from the abstract Player class.
- ComputerPlayer – Game : A Game might have a ComputerPlayer is a user chooses to play with a bot.

Game

Reason for choosing this domain

The Game domain defines the state of the whole game at all times. It initialises the main required elements (like board, tokens, players, etc.) during the startup of the game. The game runs for as long as there is no result (win or draw). Since the Game domain defines the initialisations of certain main elements, this implies that some of these are a part of the game itself and cannot exist on their own without the presence of a running Game. Hence, a composition identifier is used for the relationships between Game and these elements because none of these elements can exist without a running Game. Even the Player domains will not be able to act unless there's a running Game, hence the Game is composed of the Player as well (especially ComputerPlayer).

Reason for the relationships

- Game – Player: Every game has two players.
- Game – MessageBox: During the lifetime of a game, multiple messages may be displayed to inform the players the state of the game. Composition is not used here because MessageBox is not a part that makes up the Game, it is merely contained by the Game.
- Game – Board: Every game is comprised of a game board
- Game – Token: Every game is comprised of 18 tokens
- Game – Display: The state of the game at every point is presented on the display.

MessageBox

Reason for choosing this domain

A message box is mainly required for 2 functions. Message box is displayed when a mill is formed by either of the players as an indication for the player to remove a token from the opponent. Other than that, a message box is displayed when the game ends to display the result of the game.

Reason for the relationships

- Game – MessageBox : The Game will display a minimum of 8 messages (7 mills minimum to win a game formed by the same player + 1 winning message).

Token

<u>Reason for choosing this domain</u>
Token is the main element in the game for the players to move. Actions can be made on these tokens by the respective player. The combinations of these tokens form the basis of the game's actions and result.

<u>Reason for the relationships</u>
- Token – Position: Each Token can either have 0 Position (Removed or yet to be placed) or 1 Position (Placed on the board)

MovablePositionFinder

<u>Reason for choosing this domain</u>
MovablePositionFinder is used to locate the empty positions that a selected token can be moved to. This is an interface which defines the behaviours that are used to look for available empty positions on the board. The implementation details of these methods depend on the type of Move selected. Only the types of moves which involve moving a token to a new position will implement this interface. For example, the FlyingMove will implement this interface and set up the behaviours to look for available positions along the entire board, while the SlidingMove will set up the behaviours of this interface to look for available positions along the current row.

<u>Discarded Alternatives</u>: The required behaviours for this interface cannot be included as abstract methods in the Move parent class because not all Move subclasses require these behaviours (eg: RemoveMove). The behaviours are defined in an interface as abstract methods to enable reusability and extensibility (DRY and Open-Closed Principle). Methods cannot be directly implemented in the respective Move subclasses due DRY violation, and because each Move subclass has its own implementation of these methods.

<u>Reason for the relationships</u>
- MovablePositionFinder – Move subclasses: The respective Move subclasses implement this interface because all of these types of Moves (placing tokens, sliding tokens, flying tokens) require looking for empty positions on the board.
- MovablePositionFinder – Position: The behaviours of the interface iterate through the selective locations based on the type of move to return available empty positions.

Position

<u>Reason for choosing this domain</u>
Position is required in the game to form the basic design structure of 9 Men's Morris's board so that tokens can be placed on them.

<u>Reason for the relationships</u>
- No outgoing relationship

Board

<u>Reason for choosing this domain</u>
The Board consists of positions along which all the tokens can be moved by the players. This domain defines the base location where all the actions of the game take place.

<u>Reason for the relationships</u>

- Board – Position : The Board has 24 Position where a token could be placed. Board and Position relationship is chosen to be represented as Composition since Position is part of the Board. Without the Board, the Position will not even exist as they are supposed to be placed on the Board at specific locations according to the 9 Men's Morris's board design.

Display

<u>Reason for choosing this domain</u>
The Display is used to show the current state of the game at every point.

<u>Reason for the relationships</u>

- No outgoing relationship

Move

<u>Reason for choosing this domain</u>
The Move domain represents an abstract class which would define the basic action for all the various types moves a player can make. All the types of moves involve changing the position of a particular token in the game. The position could also be out of the board, like in the case of RemoveMove. Since, all the move classes have the same basic action of shifting a token from one state to another, all of them extend from the Move super class. The Move class contains the necessary behaviours required to shift the token's state, hence it can be commonly accessed by all subclasses. This prevents violation of the DRY principle and allows for new types of moves to be added in the future without significant changes to the design.

<u>Discarded Alternative</u>: The Move class may not consist of too many methods and attributes since it only defines the basic shifting position action, while the actual action is still individually defined by each Move subclass. Hence, an alternative to this design could be having these limited attributes and methods in every individual Move class. This alternative was discarded because it was deemed unnecessary to contain repeated code elements when it could just be reused. Repetition would make each move class more complicated and the design would be harder to extend in the future. Additionally, it is always a possibility that the Move class could have added functionality common to all types of moves if the requirements change in the future, hence, having a separate Move abstract class was a better option.

<u>Reason for the relationships</u>

- Move – MillChecker : After every move of sliding, placing, and flying, the MillChecker checks if a mill has been formed. In the case of removing, it is used to ensure that the token removed by the player is not part of a mill. Since all the subclasses (PlacingMove, SlidingMove, RemoveMove, FlyingMove) of the Move will eventually have to use the same implementation of MillChecker, the MillChecker has a direct dependency with the abstract class Move instead of all the individual subclasses.

- Move – Position : Move updates the Position of a token when it is repositioned.
- Move – Token : Move changes the positions of Token

SlidingMove

Reason for choosing this domain

This move provides a way for tokens to move to adjacent positions, which is the basis for the game itself. SlidingMove is only available to the players once they have placed all of their tokens on the board.

Reason for the relationships

SlidingMove – Move : SlidingMove inherits the basic position shifting behaviour from the Move parent class

FlyingMove

Reason for choosing this domain

This move provides a way for tokens to move to any unoccupied position on the board (flying). FlyingMove is enabled when players have three tokens remaining.

Reason for the relationships

FlyingMove – Move : FlyingMove inherits the basic position shifting behaviour from the Move parent class

PlacingMove

Reason for choosing this domain

This move is used by the players to place each of their 9 tokens on the board during the start of the game.

Reason for the relationships

PlacingMove – Move : PlacingMove inherits the basic position shifting behaviour from the Move parent class

RemoveMove

Reason for choosing this domain

This move is used to remove one of the opponent’s tokens from the board. A player that has formed a mill will get to remove one of the opponent’s tokens.

Reason for the relationships

RemoveMove – Move : RemoveMove inherits the basic position shifting behaviour from the Move parent class

MillChecker

Reason for choosing this domain

The MillChecker domain defines behaviours to loop through all positions on the board and find if a mill exists. After every move, these behaviours are run to check for a mill on the board. If a mill is found, a message is displayed. Depending on the type of move done, additional functionality may be added to support the MillChecker functionality. For example, for RemoveMove, the mill checking behaviours will have to be run ***before the remove action*** takes place and additional functionality will have to be added to ensure that the tokens in the mill are not removed (since tokens in a mill cannot be removed). But for other Move actions, the mill checking is needed to be done only ***after the move action*** takes place so that the updated state of the game can be displayed to the player.

Reason for the relationships

- MillChecker – Position : The MillChecker goes through all 24 positions to find a straight row of 3 occupied positions (mill) aligned along the same board line.