FIT3077: SOFTWARE ENGINEERING: ARCHITECTURE AND DESIGN

Sprint 1 (20%)

Done By

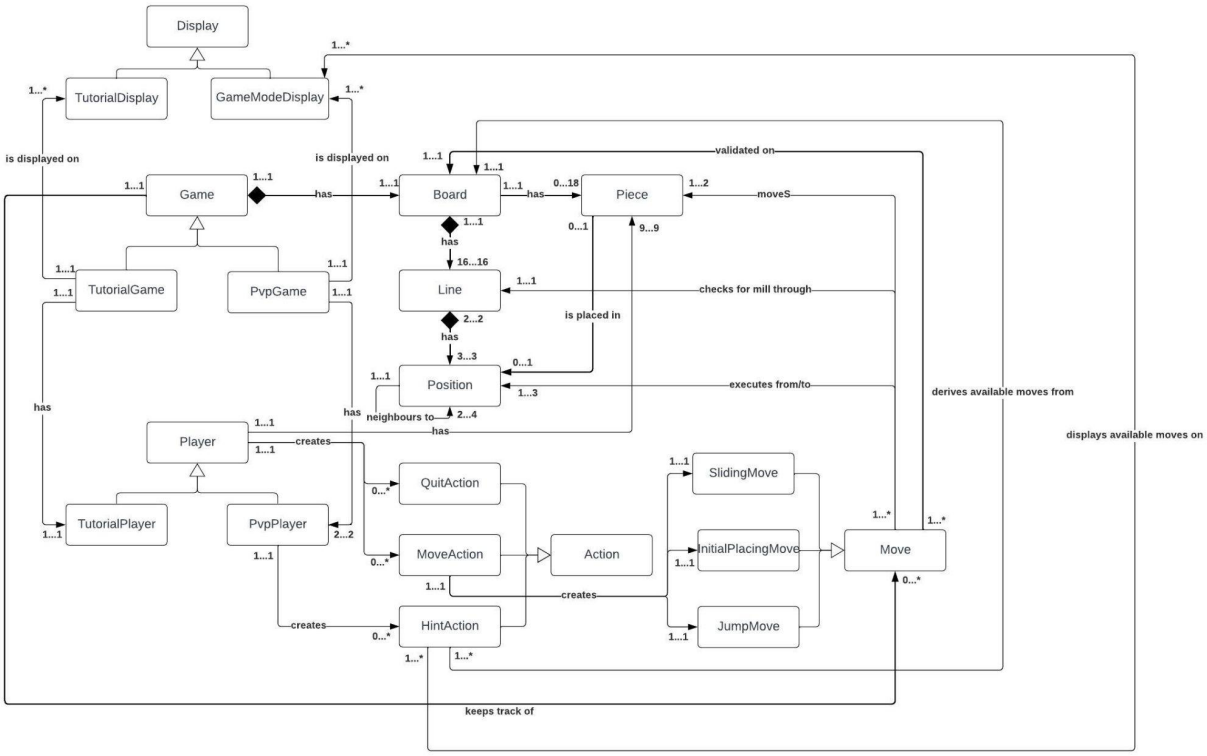
Shillong Xiao (Norman) (30257263)

Sok Ear (32442688)

Kisara Batugedara (30912539)

Faculty of Information Technology

1. Domain Model



2. Domain Entity Definitions

- **Game**
  Game is an abstract class that contains the values of the game itself. There are two types of game modes including Tutorial game and pvp game. Those two games are the child classes of the game. They inherit the properties from game class

- **Display**
  Display is an abstract class that contains fundamental information about the display of the game. It is responsible for handling all the display of gameplay. There are two types of display for the game: tutorial display and game mode display which are sub-classes that inherit properties from the display parent class.

- **Player**
  Player is an abstract class that represents the player themselves. There are two types of players including tutorial player and pvp player. Those two players are the child classes of the player class. They would inherit the properties and functionalities from the player class.

- **Board**

Board is the class that represents the board. It is responsible for storing information about which token is located at a certain position on the board.

- **Piece**
  Piece is the class that represents a single piece that a player has. It is important to make sure that every single piece is unique and to connect with other classes.

- **Action**
  Action is a class that represents an action made by a player at any point of a game. Since there are different types that can be made, the action class will act as an abstract class with multiple subclasses extending from its functionality. These subclasses are
    - QuitAction which is responsible for when the player quits the game
    - MoveAction which is responsible for when the player makes a move on one of their pieces
    - HintAction which is responsible for when the player toggles the “hint” button in the game to request hints.

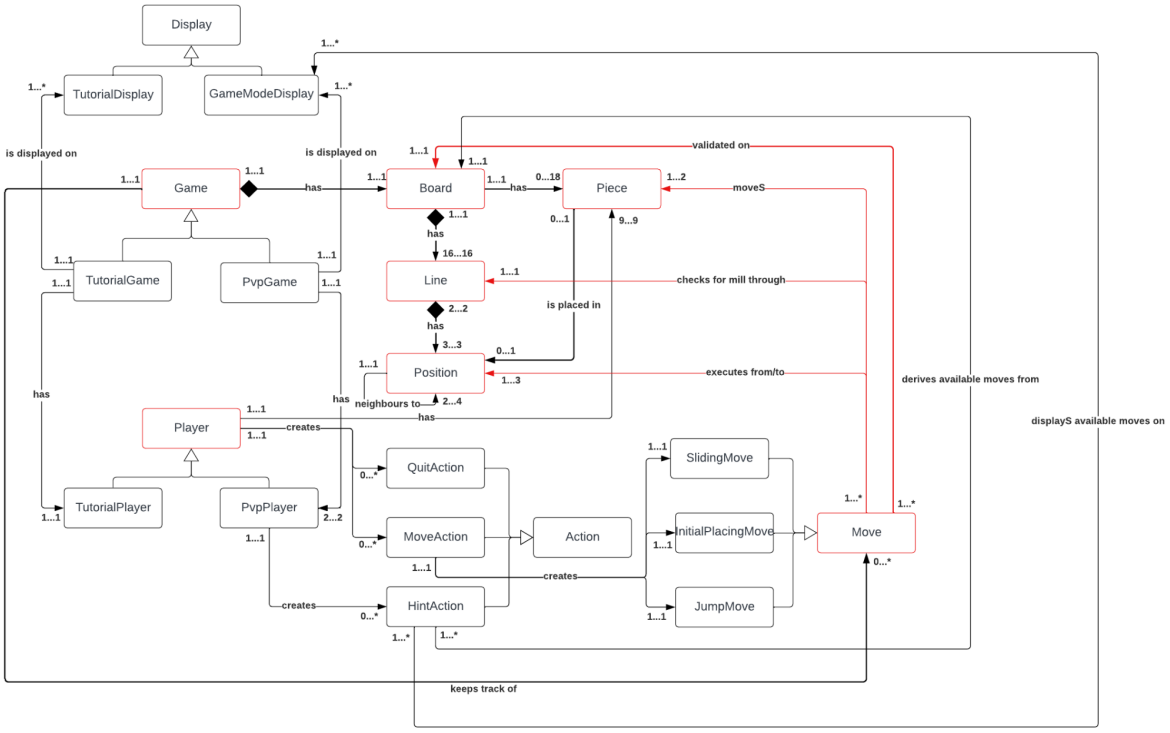
- **Position**
  A position is a class that represents an intersection of two lines on the board where the player can potentially place their token on or move/jump their token to.

- **Move**
  Move is an abstract class responsible for executing the moving of pieces from a position on the board to another position. Since there could be different types of move depending on different stages of the game, it is best to create child classes that extend from this class to reflect this. These child classes are:
    - SlidingMove
    - InitialPlacingMove
    - JumpMove

3. Functionalities rationale

3.1. Set up the board (including all pieces)

To set up the board with all the available tokens for each player, the following relationships and entities that are highlighted in red will assist with the execution of this functionality.



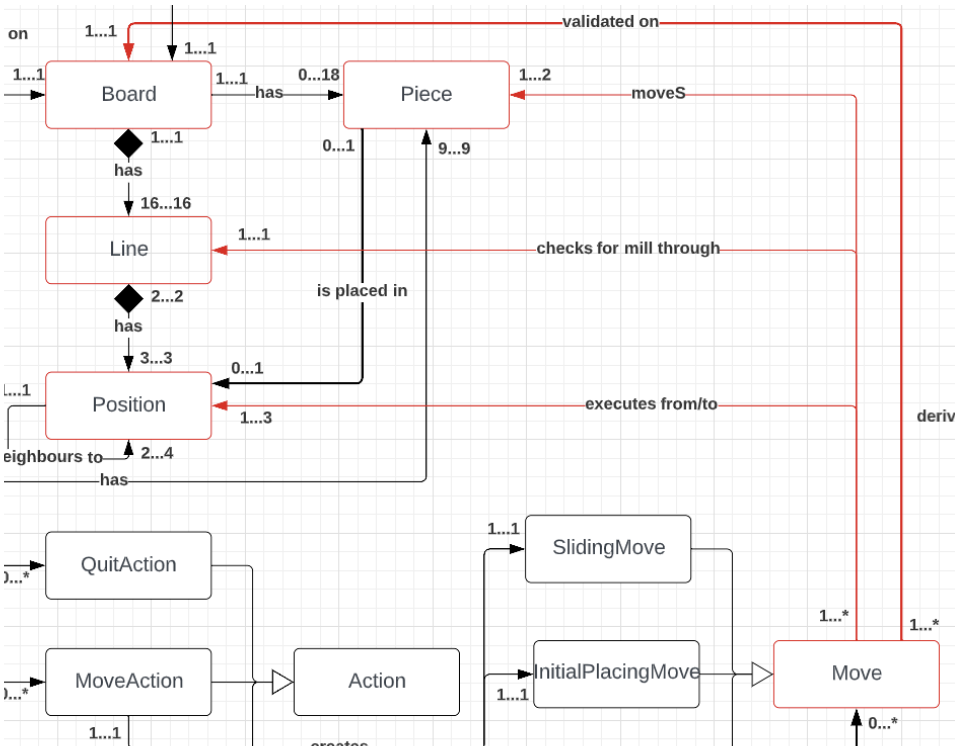
The Board class will have a direct association with the Piece class as the board contains multiple pieces (tokens). Each Piece will be placed on a certain position on the board, hence there is a direct association between Piece and Position entities. This will help the board understand the position of each token or piece that belongs to a player.

The Board is a composition of lines and lines are a composition of positions. Therefore, there is a composition relationship between Line and Board and as well as between Position and Line.

The InitialPlacingMove is in charge of facilitating the placement of tokens in the beginning of the The IntialPlacingMove is a subclass of the Move entity that has a direct association with Piece as any sort of move action needs to move a piece.

Other alternatives for this design would be to have direct associations between Board and Piece, Board and Line and Board and Positions. However, this design won't be ideal because direct associations would mean the Board would have to contain all the information about the lines and as well as the positions which would thus make it harder for us to access that information for other functionalities.

3.2. Move



The Move class's main responsibility is to execute moving pieces from one position to another. However, the nature of these moves can vary depending on the game stage, which is why the Move class is designed as an abstract class with three subclasses: SlidingMove, InitialPlacingMove, and JumpMove.

These subclasses inherit from the Move class and expand upon its functionality to handle specific types of moves. The SlidingMove subclass handles the moves where a player slides a piece from one position to another, while the InitialPlacingMove subclass handles the moves where a player initially places a piece on the board. Finally, the JumpMove subclass handles the moves where a player jumps their piece across the board.

By utilising these subclasses, the Move class can efficiently execute various types of moves in the game, making it an essential component of the game's mechanics.

Executing a move

In order to execute a move, the Move class must know these following informations:

- which pieces to move
- The position from which the piece is moved and the position to which it is moved

Thereby, an association relationship must be formed between Move → Position and Move → Piece.

Another alternative was to have a direct association between Move → Board. In this scenario, the Board would access its association relationship with Piece and its composition relationship with Line, which would then link to Position. However, getting the information will be inefficient and it can potentially lead to code smells such as Feature Envy, and consequently it is discarded.

Test for validity of move

Before the move can be executed, a validation must be made in order to ensure that the rules are followed. To conduct validation, two pieces of information must be obtained:

- The position of all pieces on the board
- The player to which each piece belongs.

Therefore, an association relationship between Move → Board must be established.

There are no feasible alternatives for this, as the Board is the only entity that can access all the information required to execute this function. This is due to its relationships which include:

- An association relationship with Player (Board → Player)
- Composition relationships: Board is composed of Line, Line is composed of Position.

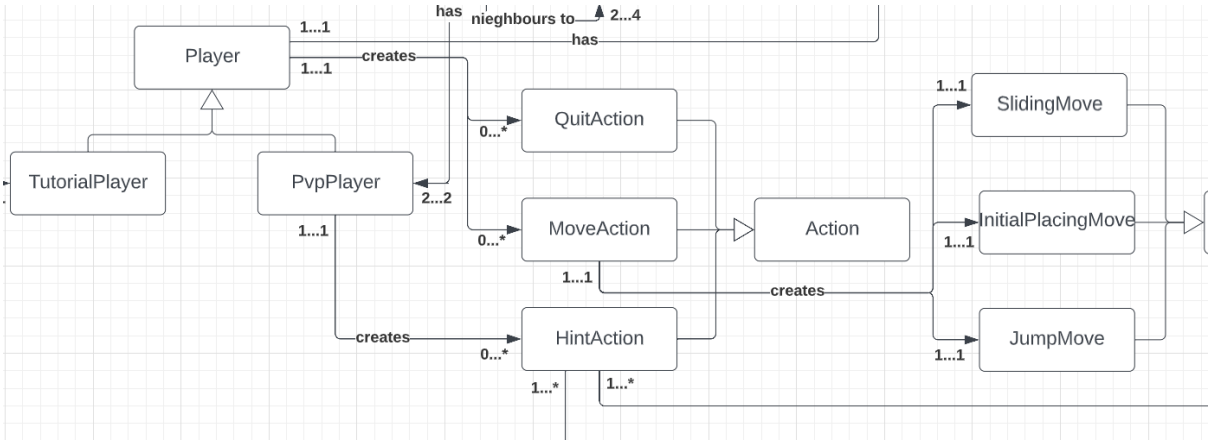
Check for mills

After a player creates a move, a check for “mill” must be conducted. This can be done by establishing a relationship between Move → Line. The Move class requires information about the line to which the piece has been moved and whether there are other pieces on that line of the same player that could form a mill.

Line is composed of three Positions, as shown in the UML diagram. Therefore, Line contains information about the three positions which allows it to determine whether each position contains pieces of the same colour (belonging to the same player), enabling it to determine if a mill has been formed.

An alternative for this is to have a direct association between Move → Board. However, this is not efficient because not all content of Board is needed to execute this function. Consequently, it is discarded for this function.

3.3. Action



As mentioned in Domain Entity Definitions, our 9MM game will have three actions that a player can implement on our board. They are:

- Quit the game
- Move the piece
- Hit the hint button.

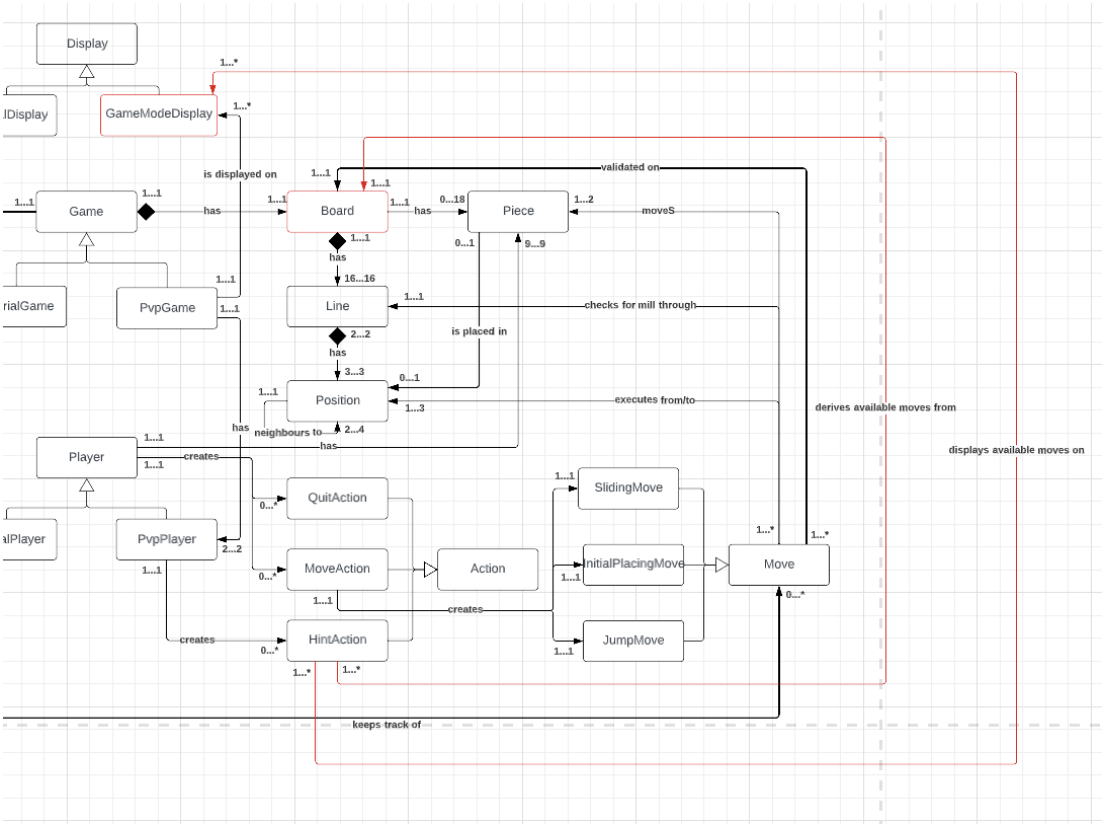
Therefore, a generalisation relationship will be placed between action entities and its subclass entities.

MoveAction will trigger when a player makes a move of their piece on their board. Depending on different stages of the game, a MoveAction can create different types of Move. Therefore, MoveAction has the association relationship with all the possible types of moves which are SlidingMove, InitialPlacingMove and JumpMove.

As mentioned above, an action is implemented by a player, so there is an association relationship between players and all the actions.

We are assuming that a player in the tutorial mode will not have a hint action since the tutorial mode is already a step-by-step guide to how to play the game. Therefore, a player entity can have the association relationship with both QuitAction and MoveAction. While HintAction will have the direct association with only pvpPlayer who is playing in PVP mode.

HintAction Assumption:



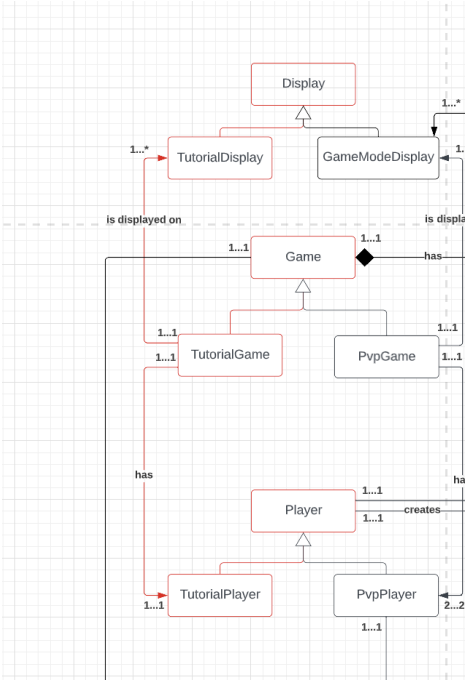
We assume that the hintAction will work as a button and when the user hits the hint button, there will be a glowing light showing the possible pieces that can move. When clicked on one of the pieces, the board will show the possible paths of the piece by glowing on the position.

In this case, HintAction should use the current display of the game to display these hints. Therefore, there is an association relationship with gameModeDisplay. These things will be derived from the board as the board will possess all the information about all the pieces, the player to which the piece belongs, and their positions on their board. Therefore a HintAction has an association relationship with the Board.

Why not the relationship between HintAction and Position?

Here, we decide to make the association between HintAction and Board instead of HintAction and Position. It is because the Board entity can access all the information about each position through its composition relationship with Line, and then from Line to Position. On the other hand, if an association relationship between HintAction and Position is established instead, 24 of those positions will be needed in order for HintAction to successfully execute its purpose.

Apply tutorial mode



Tutorial mode is the separate mode from the actual game play. It's like putting players into multiple gameplay scenarios and explaining the rules of the game by leading the player to play the game.

We planned to place a tutorial button on the homepage of the game. When the player clicks on the tutorial button, the player will be navigated to the tutorial page. Inside the tutorial page, we will set up multiple game scenarios. The game will force the player to click on the specific position on the board and interface will jump to the next scenario.

To separate the tutorial player and gamemode player, we create a tutorial display to be associated with tutorial games. Also a tutorial game has a relationship with the tutorial player.

Although both TutorialDisplay and GameModeDisplay both display the board at every turn of the game, outlined by their inheritance to Display, they perform so differently. Tutorial display will constantly show the board with written texts that explains each game scenario to the player whereas GameModeDisplay is only interested in showing the actual game board without having any texts.

‘

4. Design principles rationales

- Single responsibility (SRP) :

  When we designed our domain model, we considered SRP. As a result, each of our domain entities only has one specific and well-defined responsibility which is outlined

in section 2: Domain Entity Definition. Our adherence to SRP ensures that our program will be easy to test and maintain. This will also eliminate the possibility of future code smells such as God Class or Long Method which can reduce readability and extendibility. Overall, our commitment to the SRP can assist in achieving a more and robust system.

- Open/closed:

  We considered the Open/Closed principle when designing the domain model. Consequently, inheritance relationships are often used where possible thereby opening up opportunities for extension without having to refactor the current code base.

  For example, having Action class being an abstract class ensures that whenever we need to implement new actions, it can be extended so easily by creating a subclass of Action, without needing to modify the current code.

- Liskov substitution: (LSP)

  In all the inheritance relationships, all the subclasses will behave consistently with their superclass thereby adhering to LSP. The superclass will act as an outline of the subclass, and will define its basic properties and functionality.

  For instance, in the Player to TutorialPlayer, and Player to PvpPlayer inheritance relationships, both subclasses can perform all actions that a Player can, such as QuitAction and MoveAction, which upholds the substitution principle. This approach is applied to all abstractions used in our model, such as the Display, Action and Move abstractions, where all subclasses have at least the same properties and functionalities as their superclasses.

  Adhering to LSP ensures safe extensibility of the system in the future, making it more robust and easier to maintain.

- Don’t Repeat Yourself (DRY)

  Since the design has multiple abstractions, it allows adherence to the Don't repeat yourself principle (DRY). By having subclasses extending functionalities from its parents, it makes the code more reusable thereby reducing the need to repeat the code.