Sprint 2 Deliverable

Architecture and Design Rationales

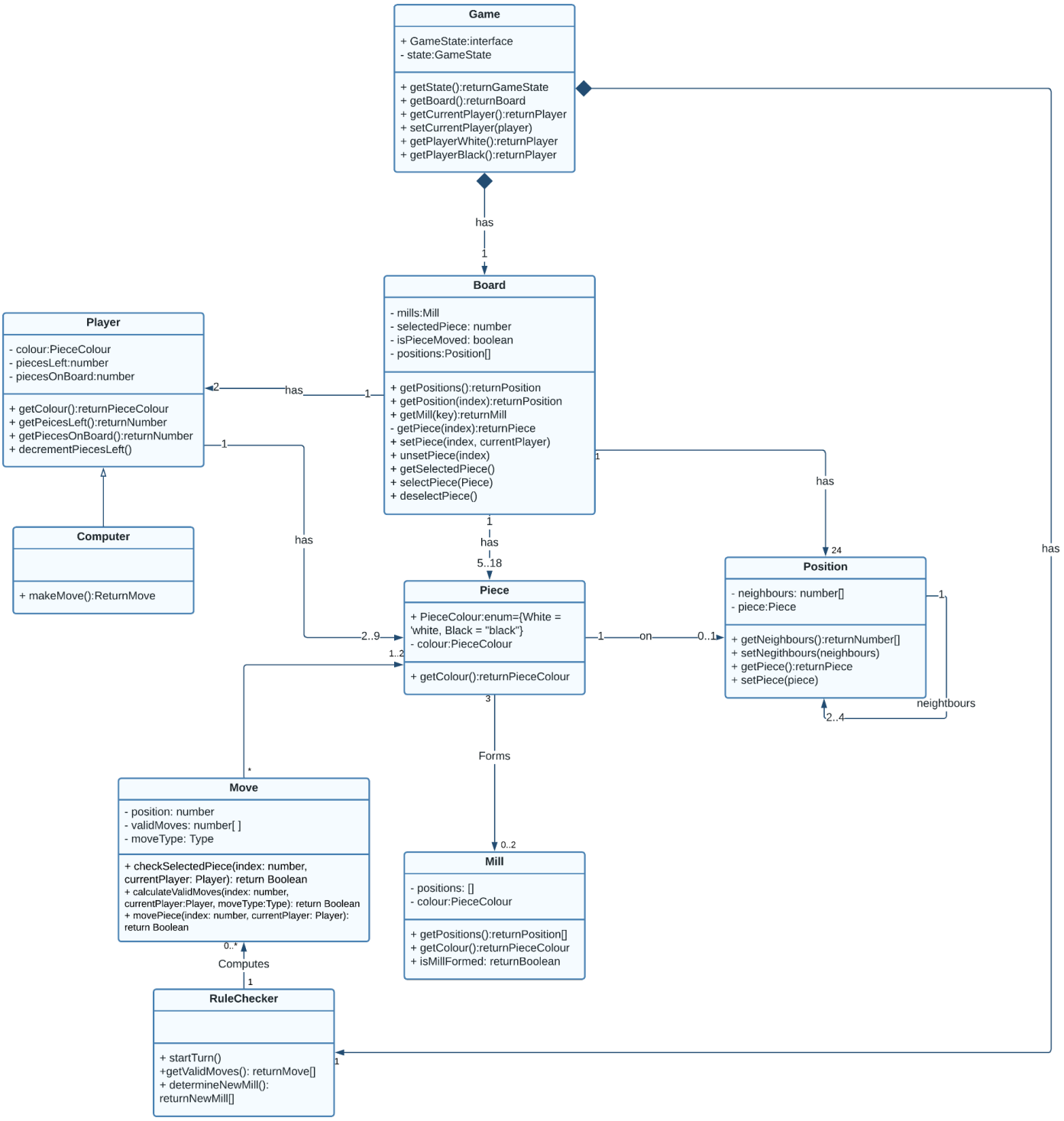
Prepared by 3 Men's Morris
Jer (Arthur) Lin, Quoc (Harry) Han and Liangdi Wang

Contents

1. Complete Class Diagram

A Complete Class Diagram for the 9MM game, including Class names, attributes, methods, relationship between classes, and cardinalities.

LucidChart: [link to class diagram](#) (also in repository /docs)



2. Design Rationales

2.1. Classes

The Game Class

The game class is required for initialisation of the game and keeps track of the state of the game until a player wins. Some of this information includes the two players, the game board. Additionally, the game class has the responsibility of switching turns and interacting with other classes such as the RuleChecker to enforce legal moves to be made by the players. The Game class is required as it acts as the game master and orchestrates the state of the game, which allows the flow of the game to be smooth through using all the available information and consolidating them into a single source of truth. As all this logic is only scalable when a clear separation of concerns is enforced, the Game class is created to avoid the single long methods code smell, which will occur if it was to perform all these described tasks.

The Board Class

The board class will encapsulate all the objects that are elements to the board. For example all the positions, pieces and mills formed. It also serves as a medium for pieces and positions to interact with each other, having all the pieces and positions consolidated together allows the application to detect and stay aware of the existence of the pieces and the occupancy of any positions. Having access to all the information in a consolidated board class allows the RuleChecker class to compute all the legal movies based on the state of the board, the state of board will be accessed by the Game class, which will allow the Game class to switch turns. The board class also has access to the two players, which will allow the two players to be distinguished by their respective colours.

The Piece Class

The game will have up to a maximum of 18 pieces. Therefore, it is not reasonable to declare 18x (number of unique state) variables, which becomes inefficient when it comes to accessing information regarding a piece. This approach, without any sort of standardisation, will make the application really difficult to remain. Hence the rationale behind this Piece class, the piece class will allow all the information regarding a piece to be captured in a single class. Some of this information includes colour and the respective methods allowing the management of the pieces.

The Position Class

The position class exists for the same reason as the piece class, as there are multiple positions on the board, the need for separation of concerns regarding each position allows for more efficient development. The positions class also allows the RuleChecker to determine legal moves as all the neighbouring positions are stored as an array and made aware of within each position object. It also serves other purposes such as occupancy, to help with the determination of any Mills on during the game. Grouping relevant details of a

piece is more efficient as it gives the application a clearer depiction of the spatial composition of the board. Which is not very easy to achieve with a single method or multiple variables

The Mill Class

The mill class serves as a template for the definitions of all the required positions to form a mill. Even though this information can be expressed in an array, the mill class will also have the ability to determine if a mill is formed, done through its methods and keep track of mills formed and the respective colour that formed the mill. This enforces the single responsibility practice, which will reduce the overhead for the board class which also abides the avoidance of having a god class.

The RuleChecker Class

The rule checker class reduces the computational overhead for the game class. Rather than having the game class compute all the valid moves for a player and switching over turn, the rule checker will perform these tasks on the game's behalf. This will allow the game to delegate all the tasks of changing the board state and determination of board condition to a single class, and only call the rule checker once to process changes. Which gives clearer separation of concerns, where the Game keeps track of the state of the board while the rule checker keeps track of patterns and changes of the board.

2.2. Relationships

The Board class is a composition of the Game class. As the Game class represents the game and initialises the board, the board is a composition. As the game is deleted or ends, the board will not exist either. Due to this reason, the Board class is a composition as opposed to an aggregation.

The RuleChecker class is a composition of a Game class. This is because the RuleChecker facilitates the movement of the game. If the game does not exist, the RuleChecker will not have access to any game state and will not serve any functionality, so it will not exist either. Therefore, RuleChecker is a composition too

2.3. Inheritance

Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?

The Player Class

In 9MM, we will have two players that have the same way of behaving throughout the game, - including placing, flying, and moving their pieces, as well as aiming to win. Hence, we don't

need to use the inheritance technique to create a different inherited class of Player class for these players since the only difference between them is the colour attribute.

The Move Class

The Move Class can be used to calculate the set of valid moves for the players each turn. The Move Class will have some attributes such as current position of piece, what type of moves the Piece performs and the set of positions the Piece can move to. Hence, we don't need to create the inherited class for each type of move in 9MM (Place, Fly or Adjacent Move) since these moves don't have any special attributes or behaviours. The only difference between them is the condition they could be performed by the player and the condition can simply be kept tracked by the Board entity.

The Fly Class

Although the Fly move is one of the three moves the player can perform in 9MM, we don't consider the Fly move as the inherited class of Move class in our design because Fly entity doesn't have any special attributes or behaviours that can be standalone. The Fly move should be considered as one of the move behaviours in Board class since it can only be used when players reach the condition that they only have two pieces left on the board.

The Place Class

Place move is nothing more than a Fly move or vice versa. At the beginning of the game, every player needs to place all of their 9 pieces to the empty position in the board and they can place them randomly or strategically. Hence, if we consider Place move as a Class, their usability will not be efficient since they don't have any special attribute or behaviour that the Move can not have like validating the piece's move or moving the piece. It would be optimised if we consider the Place move as one of the behaviours of the Move class.

The Remove Class

Removing the piece is more like a task than an entity. Thus, we definitely don't need to use the inheritance technique to implement the Remove since the piece can be removed by the Board as the Board entity keeps track of all the Piece entities placed on them and the Board entity can also check whether the Piece entity needs to be removed (or we can say that the Board entity set the position of removed Piece to null according to our design).

2.4. Cardinalities

Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1...2?

Some cardinalities are documented in details as per the following:

- Every game will have a have a board
- Every Game will have a RuleChecker
- Every player has a maximum of nine pieces and a minimum of two as they will lose when there are two pieces left
- With the above logic in mind, the board will have a minimum of 5 pieces as a player can win with a minimum of only 3 pieces left + 2 from the losing player
- Each position can have up to a maximum of 4 neighbours (Up, down, left and right if they are all applicable)
- A mill by definition, is formed by 3 pieces
- There are multiple combinations of moves for pieces. Even though that number can be computed mathematically, it is beyond the scope of architecture work. Therefore, it is denoted as *

2.5. Architecture

Regarding the architecture aspect, we use an Object-Oriented Programming (OOP) approach to implement the game's flow and React for building the user interface of the game. Besides that, we also use the MVC model to separate the UI and the game's logic. In the Model folder, we store all the classes of the game such as the Board, Piece, Player, Move, and Position while the Component folder is used for storing React components such as Board, GameDisplay, Mill, Piece, PiecesLeft, and Position. We utilised React's capability to implement the game's turn and the game's response to the player's clicks on the board's dots. When a dot on the board is clicked, the Game entity is created and the Board entity is created as a result, with its Position and Mill entities as attributes. In our 9MM, each class has a specific responsibility. The Game Class will act as a liaison between the UI and the game's logic since it stores all the Player's entities, the Board entity, and the game's state. React can easily check for changes in the game by looking at the Game entity. Hence, our Game class has some methods which React can use like:

- getState(): Is used for retrieving the game's state.
- getBoard(): This function retrieves the reference to the Board entity, which is used to determine whether a piece on the board has been placed, moved, or removed.
- getCurrentPlayer(): This function retrieves the reference to the current Player entity.

The Board class is quite important for our 9MM since it keeps track of all valid positions on which players can place or move their pieces. The methods of Board Class are:

- setPiece(index: number): Is used for setting the Piece entity in the provided index position number in the board.
- unsetPiece(index: number): This method is used to remove the piece from the inputted index position on the board.
- getPosition(index: number): Retrieve the reference to the position at the inputted index.

- getPositions(): Is used for retrieving the set of index numbers representing the coordinates of the position in the board.

As mentioned earlier, the Move class is responsible for calculating the valid moves of a piece and validating the player's move. The methods of the Move class are:

- **checkSelectedPiece(index: number, currentPlayer: Player):** Is used to check whether the player selects their piece by comparing the piece's colour with the current player's colour.
- **calculateValidMove(index: number, currentPlayer: Player, moveType: Type):** Is used to determine the valid moves that the player can make during their turn based on the type of move they are allowed to make.
- **movePiece (index: number, currentPlayer: Player):** Is used to move the piece to the valid position calculated from the calculateValidMove method. However, since the player is allowed to deselect the piece, this method will also check for that situation.

Let's review the game's logic up to Sprint 2. During this sprint, we didn't use the Move class as we previously mentioned. Instead, we implemented all the methods needed for move validation and piece placement in the Board class. However, for the next sprint, we plan to move these methods to the Move class to reduce the workload for the Board entity, which already has to keep track of its dots. In terms of the visual aspects of our game's 9MM, we have successfully created the visual effect of a mill forming and added a hovering effect to the board's dots. In terms of communication with the player, we only use the console.log() to inform what has happened when the player selects, deselects or moves the piece.

During this sprint, the game will have two stages:

1. In the first stage, each player will first be checked whether they have placed all their pieces on the board or not. If yes, each player will take turns placing their pieces on any unoccupied dot on the board. If both players finish placing all of their 9 pieces, the game will move to the next stage.

2. In the second stage, the player will start moving their piece to its adjacent position. If the player clicks on their piece at the position that all of their adjacent positions have been occupied by other pieces regardless of colour, the browser's console will inform the player to choose their other pieces. After selecting the piece, the player can choose where to place the piece by clicking on the valid dot or deselect the piece by clicking again on the piece. However, if the player already selected the piece, they have to deselect the piece before choosing the other piece.

3. Meetings

20/04/2023, 6:00PM - 6:30PM

Location: In Person

Attendees: Liangdi, Harry (Quoc), Arthur (Jer)

Discussion: Initial work breakdown, understanding requirements.

22/04/2023, 7:30PM - 8:00PM

Location: Online via Discord

Attendees: Liangdi, Harry (Quoc), Arthur (Jer)

Discussion: What tasks do we still need to do, who will do what task and when.

25/04/2023, 11:00AM - 12:00PM

Location: Online via Discord

Attendees: Liangdi, Arthur (Jer)

Discussion: Explaining code, discussing design, review feedback.

25/04/2023, 4:30PM - 5:30PM

Location: Online via Discord

Attendees: Liangdi, Harry (Quoc), Arthur (Jer)

Discussion: Explaining code, discussing design, review feedback. Explain dev setup.

25/04/2023, 8:30PM - 9:00PM

Location: Online via Discord

Attendees: Liangdi, Harry (Quoc)

Discussion: Talk about code implementation, possible strategies, test code.