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.

2. Design Rationales

(explaining the 'Whys')

2.1. Classes

Explain two key classes that you have debated as a team, e.g. why was it decided to be made into a class? Why was it not appropriate to be a method?

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

Explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?

The Board class is a composition of the Game class

The RuleChecker class is composition of a Game class

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

The Computer Class

The Move Class

The Fly Class

The Remove Class

The Place Class

2.4. Cardinalities

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

2.5. Architecture

Explain why you have applied a particular design pattern for your software architecture? Explain why you ruled out two other feasible alternatives?

3. References

[1] Reference: https://www.figma.com/