FIT3077

Software Architecture & Design

Composite

Basic Architecture

Zoe Pei Ee, Low*
31989985
zlow0011@student.monash.edu

Ci Leong, Ong*
31835996
cong0017@student.monash.edu

Guangxing, Zhu*
32597517
gzhu0009@student.monash.edu

*Equal contribution. Listing order is random.

Chosen Advanced Requirement

Players are allowed to undo their last move and the game client should support the undoing of moves until there are no more previous moves available. The game client also needs to be able to support saving the state of the currently active game, and be able to fully reload any previously saved game(s). The game state must be stored as a simple text file where each line in the text file represents the current state of the board and stores information about the previously made move. It is anticipated that different file formats will be required in the future so any design decisions should explicitly factor this in.

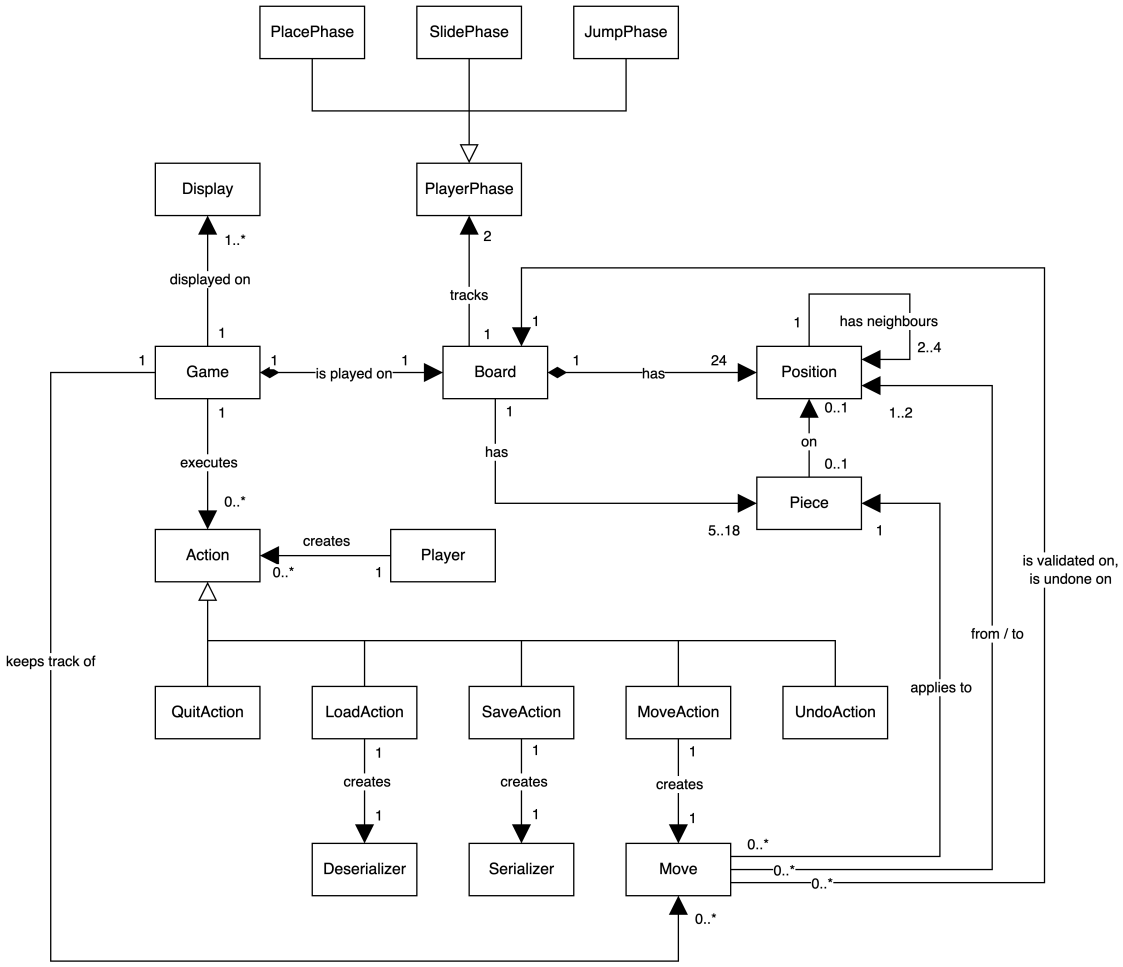

Figure 1: *Domain model*. It captures all the entities and the relationships between them for both the basic Nine Men's Morris gameplay and the chosen advanced requirements.

1 Design Rationale

This section will outline the rationale for each chosen domain and their relationships, the design choices we had to made in modelling the problem domain and why.

1.1 Entities

1.1.1 Player and Action

The Player entity represents an individual participating in the Nine Men's Morris game.

Players do not directly manipulate the game state, but instead communicate their intentions through spawning Actions. This is analogous to the way players interact with console games through a controller by pressing buttons rather than directly changing the game's state from within. This way, the Game can focus on executing the Actions without considering the specific identity of the Players. The Game only needs to distinguish between the two players and the Action is responsible for keeping track of which Player initiated it.

The primary responsibility of the Player is to interact with the Game by submitting Actions to be carried out. There are 5 types, or generalisations of Actions that could be created by the Player:

- *QuitAction* - The player submits this Action when they wish to end the current Game.
- *LoadAction* - The player submits this Action when they wish to load a Game. In executing this Action, a Deserialiser will be spawned. The Deserialiser is responsible for converting a stream back to a Game. The game will continue from there. The Deserialiser is a generalised entity. For example, the Game could be loaded from a simple text file (TextFileDeserialiser) or from a JSON file (JsonDeserialiser).
- *SaveAction* - The player submits this Action when they wish to save the current active Game. In executing this Action. In execution, a Serialiser will be spawned and it is responsible for converting the current game state, i.e. the current board configuration and the series of moves played (in order) into a stream. It is also generic - the Game could be saved to a simple text file (TextFileSerialiser) or to a JSON file (JsonSerialiser), for example.
- *MoveAction* - The player submits this Action when they wish to move one of their pieces on the board or take an opponent's piece. The Move is always applied to a Piece (see Piece), and will move the Piece from a Position X to Y. The validation is done on the Board (see Board).
- *UndoAction* - The player submits this Action when they wish to undo the previous Move they have made. It is the Move's responsibility to know how to undo themselves; the most recent Move will undo itself on the Board.

The Player entity is also decoupled from the other entities for generalisation of the Player to be introduced. For example, there could be a HumanPlayer and an AiPlayer, each capable of submitting Actions based on the current game state they see, but they achieve this task with different behaviour. Do take note that implementing a bot player is not our chosen advanced requirement.

Keeping the Action generic allows for specific player actions to be added later as a specialisation, and the Game is kept generic without knowing any of the logic how the Action will be carried out. It just needs to execute it - no questions asked.

1.1.2 Display

The Display is responsible for displaying the current state of the game for the players to see, and to react by submitting Actions.

The Display is nicely decoupled from the rest of the game as it does not need to interfere with the game logic. The computations that go inside the game engine are irrelevant to it. The Game will tell it when it needs to perform an update on the UI side.

Generalisations can also be added to the Display. For example, the Display can be a terminal GUI, or even as sophisticated as a 3D GUI.

1.1.3 Game

From looking at the diagram and reading the sections above, you should have a general understanding of the architecture. The Game represents the game engine, it orchestrates all the operations that needed to be performed for the game to be played. To reiterate, it takes inputs from the user in the form of Actions, executes these Actions, and updates the Display whenever needed to ensure a smooth gameplay.

The only responsibilities the Game handles are:

- Decides whether to execute the Action submitted by the Player, usually by asking the Board.
- Ensures the Moves are played out by each Player alternatingly.
- Tells the Display to update itself with the latest game state whenever needed.
- Keeping track of a series of Moves played out in order (to be serialised together with the Board configuration, but again, not its responsibility).

Notice that it is generic and extremely decoupled. It is unaware of the actual operations that go under the hood - it merely orchestrates them. This means we could swap out the Nine Men's Morris game with another board game that has a similar structure, and the Game engine could still be used to orchestrate the operations for that game.

1.1.4 Board, PlayerPhase, Position and Piece

The Board is what enforces the rules on how the game should be played out on the Board. In our context, it represents the Nine Men's Morris game board. If needed, it could also be converted to a generic and be swapped out with other games' boards (see Game).

It holds several key information: the 24 Positions on the board, the 18 Pieces that could be moved around the board, and two PlayerPhases, each representing a player's current phase in the game. The two players do not share the same PlayerPhase.

- *PlayerPhase* - PlayerPhase functions like a finite automaton. It has an initial state, and can move between states depending on a condition. In Nine Men's Morris, each player starts out with the PlacePhase, of which each player takes turns to place their pieces on the board, until all 18 pieces are placed on the board. Then, the PlayerPhase would transition from the PlacePhase to the SlidePhase, where players slide their Pieces along a line to another vacant Position on the Board. Finally, when a player's number of pieces fall below 3, the SlidePhase notifies the Board to transition to the JumpPhase, and the player can jump their Pieces to whichever vacant Positions on the Board he wants. When the Board validates whether a Move is valid, it will consult the PlayerPhase to test whether the Move is valid for the current game phase for that player. This model allows more game phases to be added in the future, and even reordering the phases, if the development team decides to extend the classic Nine Men's Morris rules. The reason we do not adopt the Strategy pattern approach discussed in the workshop for modelling chess moves is, in chess, each Piece has a unique behaviour. However, in Nine Men's Morris, all Pieces of the same player will share the same behaviour. It makes more sense to abstract it this way.
- *Position* - The Position entity represents an occupiable spot on the Board. Additionally, each Position is linked to its neighbouring 2-4 Positions, and this gives the Board a structure. It is otherwise structureless and there is connection between the Positions.

- *Piece* - The Piece represents a piece (or man) owned by a player. The Move is responsible for recording from which Position, to which Position, a Piece should be moved. It is also responsible for reversing this procedure. However, it is the Board's responsibility to decide whether the Move should be allowed. If the Move is allowed, it is executed and tracked by the Game.

2 Discarded Alternatives

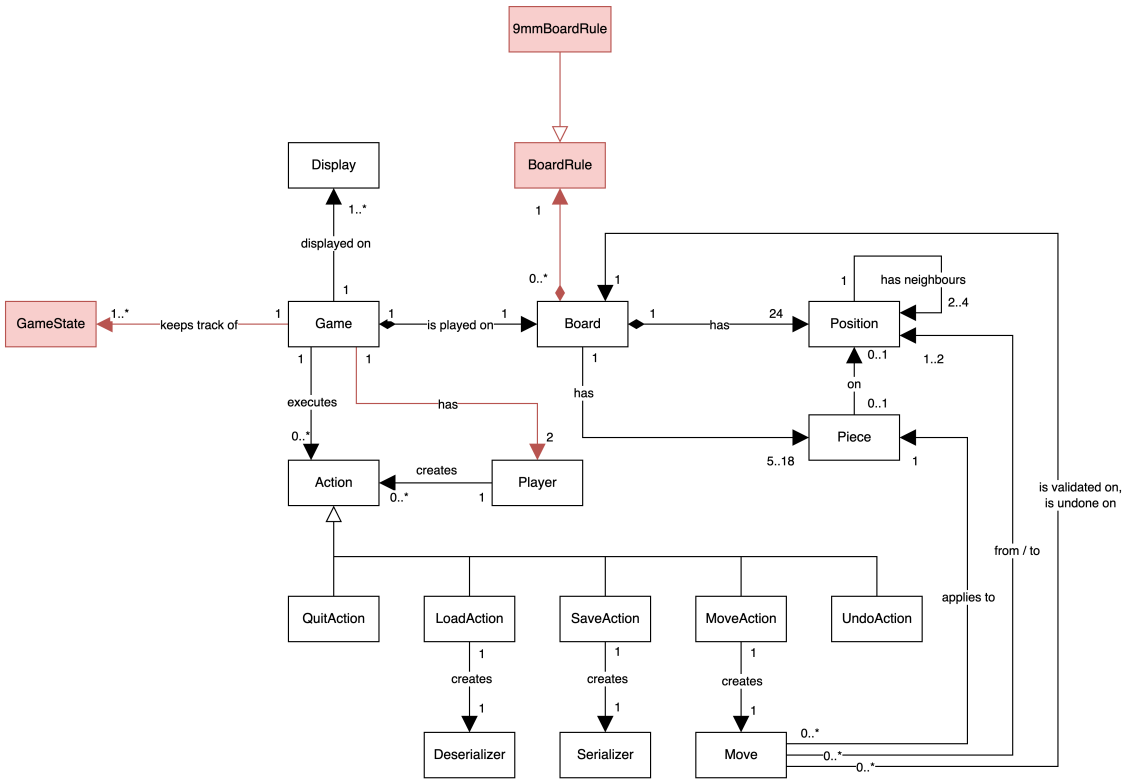

Figure 2: *Discarded alternatives*. This is the previous iteration of our current domain model. Highlighted in red are the discarded entities and/or relationships.

2.1 GameState Entity

Initially, we considered modeling the game save and load requirement by only serialising and deserialising the current GameState, that is only the current Board configuration, the current turn, and other necessary information. However, this model does not allow the loaded game to undo actions until no previous moves remain. To resolve this issue, we had to abandon this design and chose to track all executed Moves, rather than just the latest GameState. As the Moves have also been abstracted into entities adhering to the Command pattern, they can be serialised along with the board configuration at the time of saving to represent a complete game. Loading this game file will allow undoing of moves until no more previous moves are available.

2.2 BoardRule and 9mmBoardRule Entities

In our previous model, a BoardRule entity was held by the Board, which was responsible for all rule checking. Whenever a Move needed to be validated, the Board will delegate the task to the BoardRule entity. After executing the Move, the BoardRule entity is responsible for determining if a mill had formed. In each round, the BoardRule is used to check if any player had won.

This design is modular enough to allow the Board and BoardRule to be swapped with another game's, such as Chess and ChessBoard for the Chess game, while still functioning within the Game entity.

However, this approach had some drawbacks. The Piece and Position entities served no purpose and the model does not display distributed intelligence. On the contrary, the BoardRule handles everything, which breaks the Single Responsibility Principle (SRP). Furthermore, the BoardRule had to rely on case analysis for condition testing, violating the Open-Closed Principle (OCP).

2.3 Game $\rightarrow$ Player Relationship

In the early stages, the team included a Game $\rightarrow$ Player relationship to enforce that the Nine Men's Morris game must be played by two players. This is similar to how the teaching team modelled the Chess game.

However, we eventually discarded this relationship, as the number of players is irrelevant to the Game. For instance, two people could play as a single 'player' in the game by sharing the 'controller'. The important thing is that there can only be two 'controllers' sending input (in our case, the Action entity) to the game. As long as this condition holds true, the Nine Men's Morris game can be played without errors. The game does not need to know the specifics of the players - it only needs to distinguish between the inputs submitted by the controllers, i.e. Actions. The Actions already carry this information with them.