FIT3077
Software Architecture & Design

Composite

Basic Architecture

Key Classes

PlayerPhase

In our initial design, we had all of the Move validation logic kept within the Board. The validation logic would have been stored in a method or multiple methods. This however creates a monolith in our design and the set of rules to be implemented for a specific player will have to be implemented in explicit case analysis. This means none of the methods related to the Move validation logic can be closed for modification, directly violating the Open-Closed Principle (OCP).

We revised this design decision in a meeting of Sprint 2 to consider an implicit case analysis approach through polymorphism. After a long discussion, we reached a consensus that the logic should be distributed across entities (now classes) to achieve distributed intelligence for extensivity. Also, in the initial design, the Players know nothing and do nothing, and they are removed from the entity model entirely and only interact with the game through Actions. We debated and came to the conclusion that Players should still exist as an entity in the game engine as a “concept”. They should know which phase in the game they are currently in (PlayerPhase), and make the valid moves that correspond to that. This is not the game’s phase, as one of the Players could be in the phase where their Pieces could move globally on the Board, while the other Player still could only slide the Pieces along the lines. This is not the piece’s phase as all of the Pieces of a Player will share the behaviour as well.

The revised model exploits the State Pattern. The Game holds reference to two Players, and each Player holds their respective PlayerPhase, a generalisation of multiple phases in the game. Each Player is instantiated with a PlayerPhase. All Move validation logic that requires the Player’s involvement will be delegated to the PlayerPhase object at runtime. When the Player has to transition from one phase to the other, the PlayerPhase object will replace the PlayerPhase object of the Player reference it is holding (which is itself) with the next appropriate phase, through a mutator.

This makes the implementation flexible, allowing for phase transitions to be done in a fashion similar to the Finite Automaton (FA). The game engine (the core classes in the edu.monash.game package) can now be extended to implement other turn-based board games as well, as it allows for the implementation of games with non-linear phase transitioning patterns. For example, in another game, a Player’s phase transitioning pattern could be A -> B -> A -> C instead of A -> B -> C.

Mill checking logic

In our design, the Board structure is enforced by the Position class - each Position knows up to 4 Positions it is neighbours to. We debated whether to create a Mill or Row class to represent a row/column in the Board or not. However, doing so would duplicate the Board structure logic in both the Row and the Position classes.

We proposed a double dispatch approach that elegantly eliminated this issue.

In the proposed implementation, we mark the Position in the centre of each row and column in the Board as “anchors”. They can conveniently test whether themselves are part of a mill by checking whether the Pieces placed on their left and right neighbours (for horizontal mills), up and down neighbours (for vertical mills) are the same as the one placed on itself. For non-anchor Positions, the problem is non-trivial.

Therefore, for anchor Positions, they return to us the answer of whether they are part of a mill directly. For non-anchor Positions, they throw the question to the anchor they are neighbours to, as all non-anchor Positions must have at least one anchor neighbour, and the anchor neighbour will return us the answer.

Doing so conveniently moves the proposed Row logic to the appropriate class, Position, which enforces the structure of the Board through holding references to its neighbours. The Board is now responsible for only holding a list of Positions, and methods to interact with the Positions on itself, of which the Game can interface with. Distributed intelligence is achieved, the Board and Position class only possess one and only one responsibility, adhering to the Single Responsibility Principle, and they can be extended by the developer to expand the feature set.

Relationships

Game -> Board

Only the Game holds a direct reference to the Board, and naturally a Game only has one Board.

The Game can replace the Board instance however, contrary to what most people would think. This is because the Board knows how to initialise itself to the appropriate configuration at the start of the Game. If the Game marks the Board as final, and does not wish to replace the reference it is holding onto once initialised, then it would mean if the players wish to restart the Game, either the Game or the Board will have to reset the Board back to its initial configuration, by clearing the Pieces off the Board etc. It will be more computationally economical to recreate the entire Board instance and let it set itself up.

The Game -> Board relationship being marked as composition instead of aggregation avoids costly computations and reduces the amount of redundant logic within the model. Therefore, the Game -> Board relationship is composition, the Board is destroyed when the Game ends, and a fresh Board is configured for a new Game.

Player <-> PlayerPhase

We decided to adopt the Player <-> PlayerPhase composition relationship instead of a Player-InheritedPlayer inheritance structure.

The reason for this design is it allows for more flexibility because we can easily change the Player's behaviour based on the current phase. Inheritance in this case will unnecessarily complicate the code structure and could raise the problem where the Player should not inherit some behaviour from its parent but is forced to, violating the Liskov Substitution Principle (LSP).

It should be a composition as the PlayerPhase is only specific to a Player, and each Player has its own copy. Once the Player is destroyed, the PlayerPhase is no longer relevant to us. Therefore, the Player manages its lifecycle, and the relationship is not an aggregation.

Decisions on Inheritance

Actions

The Action interface represents a generalisation of the Actions that can be performed by the physical player (not to be confused with the Player class) through interacting with the JavaFX GUI.

It is designed to be a generalisation instead of a single concrete implementation to avoid explicit case analysis and matching which makes the “Action class” hard to be extended. It is also purposely made to not be in an inheritance structure, that is defining Action as an abstract class, as there are no functionalities that should be forced to implement by an Action.

Implementing the Action interface is analogous to signing a contract - it forces you to provide implementation on testing whether an Action can be executed, and how the Game can execute it. The design is such that the Game does not need to know what commands it is executing, whether or not it is correct, or how it should be executed - it simply needs to execute it once it asks the Action whether or not it itself is valid. If yes, then execute it.

This implementation adheres to the Command Pattern, and allows us to undo Actions by asking them to undo themselves, as the logic is stored in the Actions, not the Game. This will be considered in the upcoming iteration.

Methods cannot be serialised. A bonus given by this design is that the Action objects can be serialised. This gives us the ability to store Actions in text files, and load them later on when needed to implement our chosen advanced requirement. However, instead of saving and loading Actions, we will act on the stack of Moves tracked by the Game, as only the Moves are needed to reproduce the same Game state. We might reconsider saving Actions instead of Moves in the next iteration, depending on changes on the requirements.

Cardinalities

Player -> PlayerPhase

The Player holds exactly one reference of PlayerPhase at all times, and a PlayerPhase instance is only ever relevant to a single Player. Two Players can each have the PlayerPhases of the same type, though, for example, both players are in their initial Piece placement stage. So, Player (1) – is in → (1) PlayerPhase.

In our initial design proposed in the previous sprint, we had the Board storing the PlayerPhases of both players as the Board is the one used to be responsible for ALL Move validation logic (see the 9mmBoardRule entity proposed in the previous iteration). We revised this concept as this would make the Board a god class that is very hard to extend as all the logic is lumped together.

Even if we were to implement that, it is still technically impossible. The proposed idea, converted from the entity model to the class diagram model would be:

- The Board has two variables of type PlayerPhase: p1Phase & p2Phase.
- The PlayerPhase holds a reference to the Board that created it.

There is no way for the PlayerPhase to know which of the variables p1Phase or p2Phase to update upon phase change. This model does not appropriately map to this logic. Apart from that, the name PlayerPhase is a giveaway that it should be a concept that should be encapsulated within the Player, instead of the Board.

Therefore, we extracted this logic away from the Board, and encapsulated it within the Player. We ask the Players to validate their own Moves, and the players need to know which phase he is in to make the informed decisions. So, Player (1) – is in → (1) PlayerPhase.

Carefully examine the class diagram and you will find that the Move validation logic is now distributed to two classes, the Player and the Board. The Position in the Board is responsible for testing whether a Piece can be moved from, removed from, and placed to itself, and the Board is responsible for asking the appropriate questions to the appropriate Positions. The Player is responsible for testing whether the Moves they are making are valid based on the phase in the game they are in, and leave the rest to the Board. If both conditions say the Move is valid, the Move can be performed on the Board - the Game will allow it.

Position -> Position

Each Position holds 2-4 neighbouring Positions. This is a crucial relationship to enforce the Board structure. Without this relationship, the Board is an unstructured mess where the Positions are not connected to each other in any way.

We debated whether to keep the Board structure logic in the Board class or in the Position class, and ultimately we reached a consensus on adopting the latter. In this implementation, the Board becomes a public facing interface that holds the Positions in a list, and checks whether a Move can be executed on it. The Position focuses on enforcing the structure and the mill formation testing logic (see the “Key Classes” section).

We have also debated whether to store the neighbours as a set or a list, but in the end, we reached a consensus on storing them in 4 independent variables, where each of them could have a missing value, null. This is because the direction is important for testing whether a vertical mill or a horizontal mill is formed at the current Position. This is otherwise uncheckable if the direction information is masked out.

Therefore, each Position holds 2-4 neighbouring Positions in 4 independent private variables.

Design Patterns

Model-View-Controller (MVC) Architecture

As outlined in the previous iteration, we implemented the MVC architecture to separate the logic into three distinct, independent entities: all the entities within the edu.monash.game package representing the game Model, the JavaFX main.fxml file representing the View, and the ViewController (Controller) that acts as an intermediary between the two. This is a common design in GUI applications that is implemented in mobile and web applications to provide a clear Separation of Concerns (SoC) to reduce coupling. The merit of this design is that as long as the promises are delivered, i.e. the public facing interface of each entity works as intended, the three entities can each be replaced with an equal and the software will still work. A classic example would be swapping the UI layer, the View - main.fxml file with another GUI, or even a console-based UI. As this design choice is unchanged in this sprint, we will save the discussion on other patterns we have implemented.

PlayerPhase: State Pattern

The PlayerPhases are implemented with the State Pattern.

This design choice allows the phases the Players are in to transition in a way analogous to the Finite Automaton (FA). As PlayerPhase is a key class and is discussed extensively along with the implemented design pattern, this section focuses on outlining the benefits of the pattern for our use-case.

The benefits of using the State Pattern for implementing the PlayerPhase includes the following:

- *Non-linear phase transitions*: In 9MM, the phases are transitioned linearly. For example, (1) a Player first places all of their Pieces on the Board, (2) then the Player can slide their Pieces along the lines on the Board, (3) and finally it can move its Pieces globally on the Board when they have exactly three pieces left. However, with the State Pattern implemented, it could account for games with rules with non-linear phase transitions, e.g. 1 -> 2 -> 1 -> 2 -> 3 -> 1 … This allows the game engine to be reused for implementing a myriad of turn-based board games with different transitioning patterns.

- *Introduction of new phases*: To extend the rule set of the game already implemented with the game engine we designed (this need not be 9MM), the developer could create multiple concrete implementations of the PlayerPhase interface, and provide the logic for Move validation without modifying any of the Game, Board and Player logic - they are closed for modification. This gives a clear Separation of Concerns (SoC), and the developer can easily find out where to locate a specific logic to extend upon.

Action: Command Pattern

In our implementation, the physical players' intentions are communicated to the game engine via the creation, transportation and execution of Actions, facilitated by the ViewController that interacts with the View.

The Actions, implemented with the Command Pattern, based on the information it is created with, each know whether itself is valid and should be executed, and exactly how the execution should be carried out by the Game. The Game has no knowledge on what kind of Actions it has received, it only has to test whether it is valid, and if it is, execute it.

This implementation delegates the execution logic to the Actions and the Actions only. The Game that represents how the game is executed is rid of the actual implementations of the Actions. Developers that wish to use our game engine to implement another game, for example, Chess, can create their own set of Actions by signing the contract of Actions, and provide concrete implementations of the above.

Another merit of the design is that the Actions can be serialised and deserialised as they are now passed as objects instead of methods. Methods cannot be serialised and deserialised. We are still debating whether the Actions or the Moves should be serialised along with the Board configuration to allow restoration of the current game state at a later time and that will be decided in the next iteration.

Position: Fluent Interface Pattern

In setting the neighbours of a Position object, we implement the Fluent Interface Pattern. This is to make the tedious process of setting the neighbouring Positions more readable and expressive by chaining the method calls together.

Below is a demonstration of this implementation:

```
positionA
    .withLeftNeighbour(positionB)
    .withRightNeighbour(positionC)
    .withUpNeighbour(positionD)
    .withDownNeighbour(positionE);
```

Discarded Alternatives

Board holding the PlayerPhases of both players

We discarded our previous design where the Board holds the PlayerPhases of both players. However, as previously discussed in the “Cardinalities” section, this is technically infeasible. Therefore, we re-introduced the Player entity (now a class) that is responsible for knowing which PlayerPhase they are currently in, and is responsible for validating his own moves to ensure the Moves they made are valid based on the phase they are in.

See the “Cardinalities” section for an in-depth discussion.

Others

The aforementioned are some of the more intriguing rejected options. During the sprint, we have explored numerous alternatives and, as expected, discarded many of them, ultimately leading to the current state. In previous sections where we mentioned that a particular implementation was unsuitable and suggested another, the former represents the discarded alternative. To avoid repetition and maintain brevity, we have chosen not to reiterate those points here - there are simply too many of them.