FIT3077: Software engineering: Architecture and design

S1 2023

Monash University Malaysia

Sprint Two

Nine Men's Morris

Team The Three Tokens:

Priyesh Nilash Patel 32182058

Rachel Ng Chew Ern 31424290

Hee Zhan Zhynn 31989403

Sprint 2: Design Rationale

In the process of designing the architecture for this project, we had to weigh our alternatives from several different perspectives to ensure that we can demonstrate the highest quality of software design possible. Therefore, to shed light on our decision process, we worked together to document our design rationale and how we arrived at some of our major design decisions in this sprint.

Classes

One of the major changes that we made to last sprint's domain model was introducing a **Position class**. Previously, we intended to let **Board class** handle token positions, where position is an attribute in the Board class. However, as we are using pairs of coordinates to represent positions, we realised that it is inconvenient and messy to map Token objects to tuples. As a result, we created a Position class to store coordinate values, which is a much more organised and extensible approach compared to before. The design principle that is used here is the **Single Responsibility Principle (SRP)**. By introducing a separate Position class, the responsibility of handling and representing positions is delegated to that class, rather than being combined with the Board class which is responsible for the overall game mechanics.

Another major design decision that we'd like to highlight is the representation of Colour and GamePhase as enums. After considering the possible extensions that may be requested of our game, we decided that implementing Colour as an enum would make adding a new player and colour much easier. Rather than hardcoding colour values, we decided to implement player and token interactions around enums that represent a specific colour. Since Token is not directly linked to Player, this allows us to minimise coupling which means that changes to either class would not affect the other classes significantly. This upholds the **Open-Closed Principle (OCP)** which suggests that the class is open for extension but closed for modification.

The same approach was applied to GamePhase, where we use enums to signal the change in game phases such as placement phase to movement phase. Therefore, if we were to add another phase to the game, such as board and token customisation, we can just add a new enum that comes before placement phase, for example preparation phase, and implement accordingly. Similar to Colour enums, this upholds the **Open-Closed Principle (OCP).**

As a result of representing Colour and GamePhase as enums, the design is more extensible because adding new colours or game phases can be done by simply adding new values to the respective enum, rather than modifying the existing code.

Relationships

Moving on to the relationships, an interesting one is the dependency between **Position and GameManager**. As the Board is in charge of moving and updating token locations, GameManager will have to pass on the relevant instructions to Board when changes to token positions are detected. However, how will Board know where is the new location that the token is supposed to move to? Therefore, the token movement functions in GameManager take the new Position as an argument, so that Board knows the correct place to assign the token.

An alternative approach we considered was to add a Position attribute to GameManager which tracks the new location of any token movement, making it an association. It would provide more consistent information on the current game state as we can just query this attribute to see where the most recently placed token is, which might be helpful if we were to take the undo requirement. However, since we did not select that requirement, we decided to just leave it as a dependency for better code readability.

The second set of relationships that are of note is the composition of Position to Board and aggregation of Token to Board. We believe that the relationship between Position and Board is definitely composition, as the Positions cannot exist individually outside of the Board. However, we consider Token to be an aggregation as technically Tokens can still exist in the inventory of a Player if the Board is deleted. This is a correction to our domain model in the previous sprint, as we initially marked the relationship between Token and Board to be composition.

Inheritance

In regards to inheritance, we made use of it in the implementation of **Player class**, in which Player is an abstract class that is inherited by two children, HumanPlayer and AIPlayer. This allows **common attributes and behaviours** of a Player to be defined in the abstract Player class and inherited by the HumanPlayer and AIPlayer subclasses. As we have selected the advanced requirement that involves adding a Player vs Computer mode to the game, we believe that inheritance is crucial to reduce code repetition as there are many similarities between a human and AI player. Inheritance also facilitates the **polymorphic relationship** between classes. For example, the HumanPlayer and AIPlayer can be treated as instances of Player class.

However, it is worth noting that inheritance should be used judiciously as it can introduce **complexity and coupling between classes**. Hence to avoid overusing inheritance, we limited its usage to the Player class as all of the other modules are distinct and serve their own unique purpose. At the same time, there are no other clear abstractions of inheritance that could be justified in our model, thus they are implemented as normal classes to **balance code reusability and complexity**.

Cardinalities

Looking at our diagram, there is a cardinality set that prompted extensive discussion in our team, which is the cardinality of the association between **Token and Colour**. While it is clear that each token can only have one colour, the opposite cardinality is more ambiguous as we need to consider the issue of tokens being removed, or a possible extension of the game where extra tokens beyond the preset 18 are added.

In the end, we decided to go with 2..9, where each colour can represent from 2 up to 9 tokens. Each colour can also represent a maximum of 9 tokens, as we decided to not take into account extensions to the game as of yet.

As for the minimum, when tokens are removed from the board, we consider them to stop existing in the context of the current game, therefore the tokens that can be represented by a colour will dwindle as the game progresses. As the game ends when a player has less than 3 tokens, this means that there will be a minimum of two tokens remaining on the board, therefore it is not possible for a colour to represent less than 2 tokens in any given round. Additionally, during the placement phase, the tokens are considered to be existing in the players' inventory, so the tokens are already being represented by their specific colour in that phase. Overall, there are no situations that we could identify where a colour can represent less than two tokens.

The next cardinality that we'd like to discuss is also related to the above, which is the set of cardinalities between **Token and Board**. Since our current game only supports one round at a time, this means that only one Board will exist at any given moment, therefore the cardinality of Board to each Token is clearly 1. However, the interesting part is Token's cardinality, which is 0..18.

Initially, we wondered if the minimum number of tokens on a board should be 5, as the game ends when a player has less than 3 tokens. This means that there should be a minimum of 5 tokens on the board when a game ends, where 2 tokens belong to the defeated player and 3 belonging to the winner. However, after going through the general feedback given on Sprint 1, we realised that there is one case where a board can have 0 tokens – which is the start of the placement phase. As it is, there can also be a maximum of 18 tokens on the board, which is a possible scenario at the end of the placement phase. After considering all of the above factors, we determined that 0..18 should be the most accurate cardinality in regards to Tokens related to Board.

Design Pattern

For our project, we decided to follow the **MVC** (Model-View-Controller) design architecture as we found it to be the most intuitive. Two out of three members of our team have had experience working with MVC on industrial-scale software, thus we have a clear understanding on how MVC facilitates building complex, scalable applications. While the initial learning curve may be steep, especially for the third member who has not experienced

MVC development before, we believe that the higher initial investment is worth the strong foundation that MVC can provide for our project.

Advantages of MVC

Separation of Concerns	Due to MVC's modularisation of the project into three interconnected subcomponents, we believe that this structure will make it easier for us to implement and extend each component independently without affecting the others. The mode handles the data and business logic. The Controller is the bridge between model and view which handles user inputs and updates both other section. The View handles user interface.
Scalability	Each component can be extended or modified without affecting others which makes the application more flexibility to requirement changes.
Reusability	Any component can be reused by others as they are independent. For example, a Model class can be used in different Views.
Testability	Each component can be tested separately, making it easier to isolate and identify the problem if any.

Discussion on Alternatives

Before deciding on MVC, we considered implementing an **observer** design pattern instead of View-Controller in MVC. We believed that the observer pattern would work well to send data on user input to the backend without changing the Subject or Observer class, which is in line with the Open-Closed Principle. The flexibility offered by the feature of subscribing and unsubscribing to observers based on the needs of the individual module was a major advantage.

Despite that, after evaluating the scope of our project, we decided against using the observer design pattern as it may introduce unnecessary complexity and overheads. As our project flow is fairly straightforward, the effort spent setting up observer channels might be redundant if there is only one subject that is dependent on the state of the object.

Another design pattern that we considered is using a **factory method**, which is a creational design pattern, to instantiate the required classes to run our game. The use of a factory method allows us to create new objects wherever necessary without affecting the existing code, which reduces coupling between classes as they will interact with each other through interfaces rather than concrete implementations.

However, after a team discussion, we decided that the additional complexity to integrate a factory method into our code is not worth the benefits it brings. After all, due to the limited

project scope, most classes used in the game are only initialised in one place, therefore a factory method would only be used to generate one instance of a class throughout the whole game, which seems like a waste. As a result, we decided to keep our design of directly instantiating objects when required instead of relying on a factory method.