Architecture and Design Rationales

Design Rationales

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?

One of the classes that the team had debated was the Display class. The main functionality of the Display class is to show the UI of the application including the main menu, the list of previous games, and the game board etc.

The Display class could be seen as unnecessary if it is only responsible for showing the UI, as it can be achieved through a set of independent functions that handle the UI display of different components. If the UI display functionality is relatively simple and involves handling only a few UI elements such as a button, a set of methods may be sufficient. However, not just showing a button, the application also needs to show the board of each game and each game can contain many boards (different versions of the board). Hence, rendering the UI is quite complicated, as it involves handling multiple complex UI elements, meaning that making a Display class that is specifically responsible for rendering the UI is more scalable and organised.

There are some benefits of implementing the display functionality as a class. First, a class encapsulate the UI display functionality into a single unit, making the code more modular and maintainable. Additionally, a Display class provides a more organised and scalable approach for handling the UI display functionality. For example, different methods of the Display class can handle different aspects of the UI display, such as displaying the main menu, displaying the game board, and displaying the victory message. Display methods may spread over different class such as Game, Board and Position if we are not using a class, which is hard to manage. Moreover, the class can be easily extended to support new display requirements when adding new advance features.

Therefore, after discussion, the team decided to implement the display functionality as a class, as it provides a more organised and scalable approach for rendering the UI, making it more modular and maintainable.

The Team class is another class that we have debated. Originally, all the components inside the Team class belonged to Board. For example, the numAliveTokens attribute in Team was originally an attribute named as cat/dogTokenRemaining in Board. There were also methods in Board, such as a placeToken method which would place a team's token on the a position and decrease the team's unplaced token number if in the placing tokens phase. While implementing this original design, we noticed that there were a lot of repeated code,

complicated dependencies and unnecessary complexities to figure out the correct team's data to manipulate.

By creating a separate Team class, we can encapsulate the data and behaviours related to each team into a single unit, making the code easier to read and maintain. This was all made simpler when we implemented the Team class, which condensed these methods and attributes to only impact the relevant team. Additionally, using a separate class for each team allows us to create multiple instances of the Team class, each with its own set of attributes and behaviours, which can be useful if we want to support multiple teams in the game.

Therefore, we have decided to make Team into class.

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

The relationship between Board and Position is composition because each board is actively holding positions. If a board is removed, all of the positions that it was holding will also be removed along with it, as a position cannot exist on its own without a board to be on. Therefore, this relationship has to be composition and not aggregation.

There is also a composition relationship between Application and Display. Each application will create and store a display to allow the game UI to be shown. If an application is removed, the display will also be removed with it, since if there is no application, there is nothing to show.

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

Inheritance has been used in the Action class and its child classes.

The Action class was originally a concrete class with methods such as placeToken and removeToken for performing place and remove token actions respectively. However, after a team discussion, we realised that making the Action class an abstract class and extending it with child classes would be more scalable and maintainable in the long term.

If inheritance was not used in Action, this class would have to store attributes that are only relevant to certain actions. For instance, in order to perform a remove token action, the action instance will need to know whether the token removal is a part of moving the player's own token or removing one of the opponent's tokens. This means that the Action class

would need store a boolean attribute to check for it. However, this attribute is only relevant to token removal and not token placement.

By making the Action class an abstract parent class and creating Action child classes, it allows us to store only the relevant attributes and methods in the corresponding class. This encapsulates the data and behaviours of a particular action in one place, making them more maintainable and organised.

Besides, by having an abstract method, execute, in the abstract Action class, it means that all of its child classes will have to implement this method. This allows the different child classes, such as PlaceTokenAction and RemoveTokenAction, to have their own implementation of the execute method, which contains the relevant operations to perform the corresponding action. This reduces duplication of code and makes the code more modular.

Another advantage of using inheritance here is the flexibility of adding new actions. If new features, such as new actions in addition to placing and removing tokens, are required to be implemented in the future, we can easily create a new Action child class and implement the relevant operations inside its execute method. This way, we can simply create a new instance of the new action class and invoke the execute method on it when we want to perform the new action, just like how place and remove token actions are performed. This makes the application more extensible, as new features can be implemented without having to heavily modify the current code.

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

The cardinalities of the relationship between Game and Board is 1 to 1..*, indicating that each game should contain 1 to many boards, and each board should only be contained in 1 game.

The reason why each game should contain 1..* boards and not only 1 board is because in our desing, Board does not only represent the game board where tokens are placed but also the game state/turn. Thus, each game can contain multiple boards to represent the different game states/turns throughout the course of the game.

The cardinalities of the relationship between Position and Player is 0..9 to 0..1, indicating that each position can only be occupied by 0 or 1 player, and each player can occupy 0 to 9 positions at any given time.

The reason why each player should occupy 0..9 positions and not 0..* or 1..* or 1..9 positions is because in a game, each player only has 9 tokens, meaning that the maximum number of positions they can occupy at any given time is 9. Moreover, at the beginning of

each game before the player places any tokens on the board, the player is occupying 0 position, meaning that the minimum number of positions they can occupy is 0.

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

The Singleton design pattern has been applied to Display class. Although this design pattern can violate the Single Responsibility Principle, as the class is responsible for both creating and managing the instance of the class, it provides several advantages. Since the display is responsible for rendering the UI, we need to ensure that all instances of the Display class are referring to the same object, so that all data, such as the game states, will be displayed accurately to the users. Besides, it provides global access to the singleton instance of the Display class, which makes it easier to use the instance and render the UI from different parts of the application.

A design pattern that was ruled out was the Factory Method, which is usually used to create instances of different types of the superclass. This design pattern is not suitable for the creation of Display instances because there is only one type of Display class and it has no child classes.

The Builder design pattern was also ruled out for the Display class. One advantage of the Builder design patter is that it provides a way to create complex objects in a step-by-step manner, which could make it easier to configure the Display class with different options or settings, such as the different web pages of the application to render. However, it may not be necessary as the Display class has relatively simple configurations. Moreover, using this design pattern can require a lot more code to create and maintain the builder classes, making the application code more complex than it has to be.

After considering the advantages and disadvantages of these three design patterns, it was decided to apply the Singleton design pattern, as its advantages outweigh its disadvantages. It is also the most suitable for the creation of Display instances, as it guarantees that only one instance of the Display class exists throughout the application.

Architecture