Design Rationale

Key class

Main class & Page class

Rationale:

This class acts as the “runner” class or the entry point of the game. This class is responsible for running or executing the application. In this game, we will have multiple pages, such as the home screen page, the setup page (for player number), the game page and finally the ending page to show the winner. To display those pages, we need to have a while loop to display it on the window and wait for the user’s action (clicking or typing). Since in every state, there are different implementations and we might have complex conditional logic, it makes sense to separate these different states of the game. Because of this, the State design pattern is studied and used for this feature. It helps in encapsulating the specific behavior of each state as well as making the codebase more organized and maintainable. As a result, in the Main class, we only need to have one state attribute which stores the current state instance (in my case, the Page class refers to the game state). Then, when the application is executed, the state will change depending on what the user clicks and it eventually will run the whole game. Besides, if these states are implemented as a method in the main class, it will be hard to maintain and organized. Implementing them in methods may cause multiple condition statements as well to handle each transition. In this case, it would make the code become very complex and hard to manage. In short, having a state attribute in the Main class helps to enhance the readability, maintainability of code as well as flexibility. The separation of concern in this feature also helps to promote SRP. OCP can be achieved as well because a new state can be easily added into the game without affecting the other class.

Game class

Rationale:

This class acts as the game engine for the Fiery Dragon game part. It is responsible for managing the game logic as well as creating all the required components of the game such as the dragon piece, the chit cards, the volcanoes and the caves which will then form the complete gameboard. Inside this class, the display of the gameboard to the window is also managed. The reason of encapsulating the game logic and the display of the game in this class is because it helps to prevent unintended interactions between different components of the game. For example, it is possible to implement the movement of the dragon in the dragon class, however, this may cause tight coupling between the dragon class and the game class as well as the Land class (volcano and

cave). This is because the game logic itself and the display of the gameboard in a circle with the cave in the outer ring requires a lot of math which involved the dragon's id, dragon's current position, the position of each cave as well as the animal type of the current player's position and the uncovered chit card. As a result, encapsulating the game logic is better for understanding and maintaining the overall behavior of the game as we need to handle multiple different scenarios in the game including the case of a penalty card drawn, the matching card with the current position, the collision of players in the same position, movement beyond cave and the winning conditions. However, for better readability and maintenance of the code, each different case is broken down into smaller parts with each managed by the respective inner method. This helps to enhance flexibility and makes the unit testing to be easier.

Key Relationship

Main class is composed of the Page class (representing each state)

Rationale:

The main class is the starting point of the whole game application. All the other states (the Page class) cannot exist without the Main class. Although the execution of each state has an order, each of these states (the concrete child class of Page class) is not the composite of the previous one. This is because the changing of a current state to a next state is mainly just a transition, depending on its behavior. Each state is distinct and manages its own transitions to another state, it is like fulfilling its role and then changing it to the next appropriate state based on the overall game flow. However, the lifetime of the state is still managed by the other class. If the Main instance is closed or destroyed, all the game flow should be halted as well. Hence, it would also be appropriate to be composition instead of aggregation as aggregation would indicate that they may still work independently without the Main class working.

Game class is a composition of ChitCard, Dragon, Volcano and Cave class

Rationale:

The game class consists of ChitCard, Dragon, Volcano and Cave class. After all, this is what makes the game board complete. Without one of those components, the game cannot be executed, but if the game ends, all these classes will not be used anymore. Hence, the game is indeed managing the life cycle of all these classes. As a result, aggregation is not suitable as this would mean that even though the game is end, all these classes can still work independently, which will be weird since they will not be

used anymore unless a new game starts. In short, ChitCard, Dragon, Volcano and Cave classes are part of the game and they cannot live independently outside of the game.

Decision around Inheritance

1. Home class, Setup class, Game class and End class are inherited from the Page class. This is because they are all similar and have common attributes and methods. Each of them represents a state in the game and they need to have the ability to change the state attributes in the Main class. Besides, they also have a common method, run(), which is responsible for the execution of the program for each state. Since they have those similar attributes and method, by considering the Open-Closed Principle and DRY, it is best to have a parent class for them which contains the skeleton code. The run() method in the parent class is an abstract method, this enforces the child class to complete the implementation of the method. Having a parent class also makes the maintainability of the code easier.
2. ChitCard, Volcano and Cave classes are inherited from the abstract Card class as they all have similar attributes and required a getter method to access the attribute. These classes all consist of an animal type which is part of the feature of the game. Having a parent class can help to adhere to DRY principles, as well as making the maintainability easier, reduce code duplication and make the code cleaner.
3. Similarly, the Volcano and Cave classes are inherited from the abstract Land class. These classes belong to different types of land in the game where some is not accessible by certain dragons. Having a Land abstract class, we can simplify the code and avoid code duplication. This adheres to OCP as well as if there is different type of land added into the game, the class can be inherited from the Land class easily.
4. Button class and chit card class. Button class is a concrete class and ChitCard class is inherited from the Button class. This is because the Button class itself can be instantiated as a simple button by initializing the class with an image of a button. The button class only acts as a class which consists of the basic functionality of a button. In the case of a ChitCard, they are clickable, hence they have the basic functionality of a button. However, they can also be flipped, which makes it different with a basic button. It is possible to let the ChitCard class to contain a Button instance as an attribute, however, considering that we need to have an image for every clickable object in the game for the user to differentiate which element is clickable, it will be better to have an image attribute for every object that consists of the clickable functionality. Hence, in this case, the ChitCard class is inherited from the button class with the is_clicked method overridden.

Cardinality

1. Following the basic rules of the game, one game only has 16 chit cards, 4 caves and 24 volcanoes. If the game is designed for extensions, those numbers may change, but for now, the basic game rules are applied. Hence, the cardinality of Game and Chitcard is set to 16 as each game has exactly 16 number of chit cards. Besides, the cardinality of Game and Volcano is 24 as the game has 24 tiles (volcanoes) for the dragon to move. The cardinality of Game and Cave is 4 because even though there may be lesser player (2-4), the minimum Cave in the game is always 4. This means that there will be two caves without a dragon owner but the other dragon cannot enter the caves as well.
2. One game may consist of 2-4 players. Hence, the cardinality shows 2..4 indicating that it will have a minimum of 2 players and a maximum of 4 players.
3. One End class will consist of one and only one Dragon. This Dragon object is used to indicate the winner of the game and for each game, there can only be one winner since if there is a dragon reaches its cave, the game will end and halt.
4. One Main class will consist of one and only one PageController. The PageController is mainly used to change the state attributes in Main and hence the cardinality shows 1.
5. One PageController consists of one Main. This is used to set the state attribute of the Main and since only one Main object will be initialized for every game, the cardinality shows 1.
6. The Main class will consist of one concrete child class of the Page class. Although there are four different states in the game, only one state is executed at each time.
7. Each child class of the Page class will consist of one PageController. This is used to indicate that it is time to change the state of the game. Hence, the cardinality shows one PageController for one Page class.

Design Pattern

1. State Design Pattern
    - This design pattern allows an object to change its behavior when the internal state of it is changed.
    - In our case, the game has multiple pages such as the home page, setup page, game page and end page.
    - Each page can be thought of as a state and every page has distinct behavior or different implementation.
    - Instead of manually invoking the execution of each page, we can just have a page stored as an attribute at a time and set the state to the appropriate page when needed and execute the page.
    - By implementing this design pattern, it allows the code base to be more organized and maintainable.
    - Actual implementation
        - Main class contains the state attribute which store the current page that is going to run. The first page will be the home page.
        - In the run method of the main class, it will execute the state attribute. The run method in Home class is then executed.
        - When the Home class ends, it will create a Setup instance and change the state attribute in the Main class to the setup page. The setup page is invoked.
        - Similarly, when the setup page ends, it creates a Game instance and passes it to the state attribute. When the game page is run and end, it creates an End instance and changes the state attribute.
        - If the player decides to play again, the end page will create a home page and set the state attribute back to home again.

2. Command Design Pattern
    - Since we have different states in our game, we need to change the state attributes in main class. Instead of directly manipulating the attributes, we create a controller class to change the state.
    - This decouples the Page class and the Main class, providing flexibility and extensibility.
    - Actual implementation:
        - The abstract Page class will consist of abstract run method which should be implemented by every concrete child class of the Page class. In the implementation, it will specify the specific action of behavior of the child class. So, in the Main class, where the Page instance is stored as the state attribute, the Main class does not need to know the exact details of each action.

- Create a PageController class where this class contains the Main class as attributes. This class is basically the remote control where it changes the state attributes in the main class when the appropriate time comes.
- The concrete Page classes will store this PageController class as attributes.
- When the execution of the concrete Page class ends, it calls the PageController class to change the page states in main class.
- As a result, the page is change and the next page is executed.