Tay Ming Hui 33245711 FIT3077

Design Rationale

To create the Fiery Dragons game from scratch, the presented class diagram has a solid design integrated with various Design patterns.

Key Classes:

1. **AnimalFactory**:
    - This class follows the Factory Method pattern, responsible for creating different types of Animal objects.
    - It provides a centralized creation point for all animals needed in various game components (ChitCards, CaveCards, VolcanoCards).
    - By encapsulating object creation, the AnimalFactory decouples the client code from the concrete Animal subclasses, promoting flexibility and adherence to the Open and Closed principle.
    - This pattern promotes code reusability, extensibility, and adheres to the Open and Closed principle, as new animal types can be added without modifying existing code.
    - Without the Factory Method pattern, the client code would need to explicitly create instances of concrete Animal classes, leading to tight coupling and difficulty in introducing new animal types. The Factory Method promotes loose coupling and extensibility.
2. **BoardSetup:**
    - This class is responsible for building the main game board, including the chit card panel, volcano card panels, cave panels, and token panels.
    - It sets up the layout, positioning, and appearance of various game components on the board.
    - Creating BoardSetup as a separate class is appropriate because it follows the Single Responsibility Principle (SRP), as its sole responsibility is to set up and manage the game board and its components. It encapsulates the logic and state related to the game board setup, promoting better code organization and maintainability.
    - If the board setup logic were implemented as methods scattered across multiple classes, it would violate the principles of encapsulation and separation of concerns, making the code harder to understand, maintain, and extend. As a separate class, BoardSetup can be easily extended or modified in the future without impacting other parts of the codebase.

By creating AnimalFactory and BoardSetup as separate classes, the design adheres to core Object-Oriented Programming (OOP) principles like the Single Responsibility Principle, encapsulation, and separation of concerns. These classes have well-defined responsibilities and encapsulate related data and behavior, promoting code reusability, maintainability, and extensibility. Implementing them as methods would have led to scattered logic, tight coupling, and potential violations of design principles, making the codebase more difficult to understand and modify.

Key Relationships:

1. **Composition between ChitBoard and ChitCard, VolcanoCardBoard and VolcanoCard**:
   - o The relationship between ChitBoard and ChitCard, as well as VolcanoCardBoard and VolcanoCard, is a composition.
   - o This decision is appropriate because the cards (ChitCard and VolcanoCard) are strongly owned and tightly coupled with their respective boards (ChitBoard and VolcanoCardBoard).
   - o The cards cannot exist independently of their containing board, and their lifetimes are managed by the board objects.
   - o This composition relationship accurately represents the strong ownership and lifecycle management between the boards and cards, adherence to the principle of encapsulation.
2. **Association between Managers and Boards**:
   - o Classes like ChitCardManager and ChitBoard ,VolcanoCardManager and VolcanoCardBoard were created as separate classes, rather than methods. For example, ChitBoard and ChitCard have a 1 to 1 cardinality, indicating that one ChitBoardManager manages one ChitBoard. Similarly, one VolcanoCardManager manages four VolcanoCardBoard instances, as the volcano card is designed to be a square.
   - o By encapsulating related behaviours within classes, the design follows the principles of encapsulation and separation of concerns, promoting code organization and maintainability.
   - o The association between managers and boards allows for a more flexible and loosely coupled relationship. The managers can interact with and manage the boards without being tightly coupled to their implementation details.

Inheritance:

1. **Animal (Bat, Spider, Salamander, BabyDragon, PirateDragon)**:
   - o The use of an abstract class Animal is appropriate as it allows for defining common behavior and shared characteristics among related classes (Bat, Spider, Salamander, BabyDragon, PirateDragon).

- By inheriting from the Animal abstract class, the concrete subclasses can reuse the common functionality and properties, promoting code reuse and adherence to the Don't Repeat Yourself (DRY) principle.
- The abstract class Animal can define abstract methods or interfaces that all subclasses must implement, enforcing a common contract and ensuring consistency across different animal types.
- This decision adheres to the Open/Closed principle, as new animal types can be added by creating new subclasses of Animal without modifying the existing code, promoting extensibility and maintainability..

2. **GameComponent(Token, ChitCard, CaveCard, VolcanoCard)**:
    - The use of an abstract class GameComponent is justified because the subclasses (Token, ChitCard, CaveCard, VolcanoCard) can exist independently of the GameComponent class, even though they are part of the game's components.
    - By using an abstract class, the design follows the principle of abstraction, allowing for the definition of common interfaces or methods that all game components must implement or follow.
    - The abstract GameComponent class can define shared behavior or properties that are common to all game components, promoting code reuse and maintainability.
    - Using inheritance in this case allows for better code organization, as the common functionality is defined in a single place (the abstract class) and can be inherited by the concrete subclasses.
    - This decision adheres to the Open/Closed principle, as new game components can be added by creating new subclasses of GameComponent without modifying the existing code, promoting extensibility and flexibility.

Cardinalities:

1. **ChitBoard and ChitCard (1 to 16)**:
    - The cardinality between ChitBoard and ChitCard is 1 to 16, indicating that one ChitBoard instance will contain exactly 16 ChitCard instances.
    - This cardinality is based on the game rules, which specify that a chit board must have 16 chit cards for gameplay.
2. **Game and Token (2..4)**:
    - The cardinality between Game and Token is 2..4, meaning that a Game instance will have a minimum of 2 and a maximum of 4 Token instances.
    - This cardinality is justified by the game's requirement of having at least 2 players (and hence 2 tokens) and a maximum of 4 players (and 4 tokens).

Design Patterns:

1. **Bridge Pattern (ChitCards, VolcanoCards, ChitBoard, VolcanoCardBoard)**:
   - The Bridge pattern is applied through the Boards and its concrete implementations of each card.
   - Each class would have its single purpose to exist. For example, ChitBoard is responsible for managing the chit cards and building the panel that displays them. ChitCard encapsulates the attributes and behaviour specific to an individual chit card. The cards can be independently existing, but in addition to a Board as its container, this makes it much easier to manage the game board and track the player movement, adhering to SRP.
   - The cards can exist independently, but when combined with their respective boards, it becomes easier to manage the game board and track player movement.
   - The Bridge pattern allows for independent evolution of the abstraction (game components) and the implementation (boards), enabling easier addition or modification of either part without affecting the other.
   - Without the Bridge pattern, putting all the cards together to build the game board would risk introducing errors and make maintenance harder.
2. **Facade Pattern (Game)**:
   - This class acts as a Facade pattern, providing a simplified interface to manage the game's setup and execution.
   - It encapsulates the complexity of initializing game components, setting up the board, and handling game logic like checking for a win condition.
   - By centralizing these responsibilities, the Game class promotes loose coupling and easier maintenance, as changes to individual components do not affect the overall game management.
   - By hiding the intricate details of various game components and their interactions, the Facade pattern promotes loose coupling and easy maintenance.
   - Without the Facade pattern, the client code would need to interact with multiple components directly, leading to tight coupling and increased complexity. The Facade simplifies the interaction and decouples the subsystem from the client.
3. **Structure Pattern (Manager Classes)**
   - VolcanoCardManager, ChitCardManager, CaveCardManager they act as a wrapper class.
   - They centralize the functionality, making the code easier to understand, maintain, and modify. They encapsulate this data and provide controlled access through methods.
   - Manager classes act as a single point of contact for interacting with specific functionalities. This simplifies communication between different parts, we only

need to interact with the manager class instead of dealing with multiple lower-level components.

  - Without manager classes, modifying or extending the functionality related to specific game components (e.g., card arrangement) would require changes in multiple places, increasing the risk of unintended side effects and making the code harder to maintain.

The design patterns applied in this design are well-justified and address various software design principles and challenges:

- The Bridge pattern promotes flexibility, extensibility, and separation of concerns by decoupling abstractions from their implementations.
- The Facade pattern simplifies the interaction with complex subsystems, promoting loose coupling and easy maintenance.
- The Structure Pattern (Manager classes) centralizes functionality, encapsulates data, and provides a controlled interface, improving code organization and maintainability.

While alternative approaches, such as tightly coupling components or exposing low-level details directly, may work in some scenarios, they often lead to increased complexity, tight coupling, and reduced maintainability over time. By employing design patterns, the provided design prioritizes principles like separation of concerns, extensibility, and maintainability, resulting in a more robust and flexible codebase that can adapt to future changes and requirements more easily.

Link to Demo Video

v) winning of the game

The video is an unlisted YouTube video, please click the URL link below to watch it

https://youtu.be/KZBzZkbEyWs?si=JK_P4ou8FYmRvFRq

Instructions on how to open the jar file ON WINDOWS

1) Download the jar file
2) Make sure your local device has downloaded Java and JDK (JDK 22)
3) Double click to open the jar file
4) Or else, open command prompt
   a) cd directory to the file location
   b) type “java -jar <filename>.jar
5) The jar file should be successfully opened

Test Plan

1. Game Setup
   - Test Case: Verify the correct setup of the game board, including chit cards, volcano cards, cave cards, and player tokens.
   - Test Steps:
     1. Launch the game.
     2. Check if the chit board is correctly initialized with 16 chit cards.
     3. Verify that the volcano cards are correctly positioned and set up in a square formation.
     4. Ensure that the cave cards are properly placed at their respective positions.
     5. Check if the correct number of player tokens (2-4) are created and positioned on the board.
   - Expected Result: The game board should be correctly set up with all components in their designated positions.
2. Player Movement
   - Test Case: Test the player token movement on the board.
   - Test Steps:
     1. Start the game with a desired number of players.
     2. Flip a chit card and move the corresponding player token based on the card's movement value.
     3. Ensure that the token's position is updated correctly on the board.
     4. Repeat step 2 and 3 for different chit cards and player tokens.
   - Expected Result: Player tokens should move correctly based on the chit card values, and their positions should be accurately updated on the board.
3. Cave Card Interaction
   - Test Case: Test the interaction between player tokens and cave cards.
   - Test Steps:
     1. Start the game with a desired number of players.
     2. Move a player token to a cave card position.
     3. Verify that the player token is correctly associated with the cave card.
     4. Repeat steps 2 and 3 for different player tokens and cave cards.
   - Expected Result: Player tokens should be correctly associated with their respective cave cards when they land on a cave card position.
4. Win Condition
   - Test Case: Test the win condition for the game.
   - Test Steps:
     1. Start the game with a desired number of players.
     2. Move player tokens according to the chit card values, aiming to reach the winning position (predetermined by the game rules).
     3. Check if the game correctly detects the winning condition when a player token reaches the winning position.
     4. Verify that the game displays the appropriate win message or pop-up.
   - Expected Result: The game should correctly detect the winning condition and display the appropriate win message or pop-up when a player token reaches the winning position.

This test plan covers the basic functionality of the game, including setup, player movement, cave card interaction, win condition detection, and reset/restart. It can be extended further by adding more test cases for specific scenarios or edge cases based on the game's requirements and rules.