Design Rationale, Testcases and Executable Deliverable and Video Link

Design Rationale

CircularBoard

This class embodies the essence of the game board in Fiery Dragon, showcasing the circular layout and hosting various game elements. Its existence ensures a clear separation of concerns, adhering to the Single Responsibility Principle (SRP) by handling the drawing logic independently. By encapsulating the board's visual representation and interactions within this class, it promotes modularity and ease of maintenance. Additionally, it fosters the principle of encapsulation by abstracting away the complexities of the board's implementation details, allowing other components to interact with it through well-defined interfaces.

This class represents the game board, a complex entity with multiple attributes and behaviours. If we were to use methods instead, we would need to pass around the state of the board between different methods, which can lead to code that is hard to understand and maintain. By encapsulating the state and behaviour within a class, we can ensure that the board maintains its state across different operations and interactions, leading to cleaner and more maintainable code. Additionally, leveraging classes allows for the creation of multiple instances of CircularBoard if needed, enabling scalability and flexibility in the game's design.

Cave

The Cave class epitomizes the concept of encapsulation by encapsulating the attributes and behaviors of individual caves within the game view. Its role is pivotal in maintaining a clean and modular design, as each cave is treated as a distinct entity with its own set of properties such as position, radius, image, and border color. By representing caves as standalone objects, it facilitates easy manipulation and management, contributing to the overall clarity and organization of the codebase. Moreover, it promotes code reusability by encapsulating cave-specific functionalities within the class, ensuring consistency and coherence in cave-related operations.

Using classes to represent caves instead of methods provides a more comprehensive and extensible solution, aligning with the principles of abstraction and encapsulation. By encapsulating cave properties and behaviours within a dedicated class, it promotes code reusability and modularity, facilitating easy management and manipulation of individual caves. Furthermore, employing classes allows for the implementation of cave-specific functionalities and features, ensuring a clear separation of concerns and promoting code clarity and coherence. Similarly, the Cave class encapsulates the state and behavior of individual caves. Using methods instead of a class would mean that the state of each cave would need to be managed externally, leading to potential inconsistencies and errors. By using a class, we can ensure that each cave maintains its state and can manage its own behavior, leading to a more robust and reliable system.

Aggregation between CircularBoard and Cave

The CircularBoard class contains multiple instances of the Cave class. This is an aggregation relationship because the CircularBoard does not exclusively own the Cave instances - they can exist independently of the CircularBoard. This allows for flexibility in the design, as caves can be added or removed from the board without affecting the board itself or other caves. This relationship is conducive to promoting code modularity and flexibility, as it allows for the dynamic addition and removal of caves from the board without tightly coupling them to the CircularBoard class. By utilising aggregation, the design maintains a high degree of flexibility and scalability, enabling seamless integration of new cave instances into the game board.

Association between Display and CircularBoard

The Display class has a CircularBoard object, but it does not control its lifecycle. This is an association relationship, which allows the Display class to interact with the CircularBoard without needing to manage its creation or destruction. This separation of concerns makes the code easier to maintain and extend. It signifies that Display creates and contains a CircularBoard object. This association facilitates the construction of the game window and the inclusion of the game board within it. However, CircularBoard remains independent of Display in terms of its lifecycle management, promoting loose coupling and enhancing the overall flexibility and maintainability of the design.

Inheritance

In my design, I have employed inheritance for the DragonCard class and its subclasses (e.g., PirateCard, SalamanderCard). This decision aligns with the principles of inheritance, as it promotes code reuse and facilitates polymorphic behaviour. By defining a common superclass with shared properties and behaviours, such as coordinates and drawing logic, I can eliminate code duplication and ensure consistency across different types of DragonCards. Hence, the subclasses can inherit these common properties and behaviours and can override or extend them as needed. Additionally, inheritance promotes the principle of abstraction by enabling the creation of a hierarchy of related classes, enhancing code clarity and maintainability.

Cardinalities

1..1

The cardinality of 1..1 represents the relationship between Display and CircularBoard, indicating that each Display instance is associated with exactly one CircularBoard. This cardinality reflects the mandatory presence of a game board within the game window, ensuring a consistent and coherent user experience.

1..4

The cardinality of 1..4 denotes the relationship between CircularBoard and Cave, indicating that each CircularBoard instance may contain up to four caves. This cardinality reflects the specific design choice in Fiery Dragon, where each game board is configured to accommodate a maximum of four caves distributed across its surface. By limiting the number of caves to four per board, the design ensures a balanced and visually appealing layout while still providing sufficient variability in cave placements. Each cave remains uniquely associated with its CircularBoard instance, facilitating code modularity and scalability while adhering to the game's predefined constraints.

Design Patterns

Singleton Pattern:
As I was deciding over the Singleton pattern, I was reminded of the Single Responsibility Principle (SRP), which emphasises that a class should have only one reason to change. Implementing the Singleton pattern would add the responsibility of managing its own instance and providing global access to it, potentially violating SRP. In my design, I've strived to maintain a clear separation of concerns, with each class having a distinct responsibility, such as representing game components or managing the game display. Introducing a Singleton pattern would blur these responsibilities, complicating the codebase without offering clear benefits. Moreover, since there's no explicit requirement for a single instance of any class to be shared globally, I've opted against using the Singleton pattern to avoid unnecessary complexity. Singleton pattern can make unit testing difficult due to its global state. Each test case would not start with a "clean slate" as it would with non-singleton classes, potentially leading to errors if the state is not properly reset between tests.

Decorator Pattern:
Reflecting on the Decorator pattern, I appreciate its adherence to the Open/Closed Principle (OCP), which encourages classes to be open for extension but closed for modification. By allowing behaviour to be added dynamically at runtime, the Decorator pattern promotes code reuse and maintainability. While the Decorator pattern offers flexibility and extensibility benefits, my current design doesn't demonstrate a clear need for dynamic behaviour modification or extension. Each class in my design has well-defined behaviour encapsulated within its methods, and there are no explicit requirements for runtime modification of this behaviour. Therefore, introducing the Decorator pattern would add unnecessary complexity without clear benefits, so I've decided against its implementation to maintain simplicity.

Proxy Pattern:
Contemplating the Proxy pattern, I recognize its alignment with the Single Responsibility Principle (SRP), as it separates concerns by controlling access to an object without modifying its core functionality. This separation allows for access control, lazy initialization, or logging to be handled by a separate proxy object. In my design, there's no indication of complex access control requirements or the need for lazy initialization. Each class manages its own behaviour without requiring intermediaries to control access to it. Introducing the Proxy pattern would introduce unnecessary layers of abstraction and indirection, potentially complicating the codebase without tangible benefits. The use of a proxy object can introduce additional overhead, especially if the proxy methods simply delegate to the real object. This could negatively impact the performance of my game. Therefore, I've opted not to implement the Proxy pattern to maintain simplicity and avoid overengineering.

In summary, while these design patterns are valuable tools for solving specific problems in software design, they may not be appropriate for my current codebase due to the absence of corresponding requirements or the potential for increased complexity without clear benefits.

Testcases

Test Case ID	**Test Description/Test scenario**	**Outcome**
1	Overall Game Board Rendering	Successful
	Test that circular board is rendered properly with the correct dimensions to fit the window	Successful
	Four caves generated and placed north, south, east and west	Successful
	Circular board is divided into 24 steps evenly and accurately	Successful
	Circular board has left enough space in the middle to hold the cards that need to be flipped	Successful
	Each cave has a different coloured border to differentiate them	Successful
	Each cave hosts a corresponding-coloured dragon token to initiate the start of the game	Successful
	Each step on the circular gameboard has an image of a relevant animal from the game	Successful
	In total, 16 cards are generated in the middle and do not touch the circular gameboard itself	Successful
	Each cave is connected with an indent to the circular game board	Successful
	Images are scaled to fit inside the gameboard steps and caves	Successful
	Dragon cards are turned over so they all look the same	Successful
2	Overall Flipping Cards Functionality	Successful
	Clicking on each type of card button triggers it to be turned over	Successful

	Clicking a flipped card will turn it back over to how it was when the game started	Successful
	Cards are drawn at randomised positions in the middle of the circular board	Successful
	Each dragon card is a different type and number of animals and pirate cards are included	Successful
	Image for each dragon card is sized to fit inside the circle	Successful
	Multiple clicks on same button tested to ensure the state toggles correctly	Successful
	Tested error scenarios such as loading invalid image files and encountering IOExceptions during image loading	Successful
	No overlapping of components	Successful

Executable Deliverable

Running Executable

The executable can be run on any platform that supports Java, such as Windows, MacOS, Linux and more, as Java is platform independent, and Swing is a built-in library.

Have JDK17.0.10 installed the platform and either double click the sprint1.jar file to run or in terminal give command of java -jar sprint1.jar after navigation of the directory of the JAR file (through cd) is completed.

Building Executable

The executable can be built from the source code by using Intellij. When in Intellij, navigate to File -> Project Structure -> Artifacts. Click the 'plus' button to add artifact through Add -> JAR -> From module with dependencies. Specify the main class to be Launcher of main.view and click option for 'copy to output directory and link via manifest. Then 'Apply' and click 'OK'. Then from the menu bar, click Build -> Build Artifacts -> sprint1:jar -> Build. Now the artifact should be in the out folder, specifically in an artifact folder, from when you will have the .JAR file. This file can be run by right-clicking it and then clicking 'Run' or you can run it from the terminal through java -jar <jar_file_name>.jar as well.

Video Link

Link: https://youtu.be/6bknuaWBP5Y

Description of File Format: The link is to a YouTube unlisted video which was uploaded in mp4.