MONASH University

SPRINT 2 - DESIGN RATIONALE

1. Design Choice

1.1 Current Design and Used Patterns

The current design of the "Fiery Dragons" game utilizes the Factory and Singleton design patterns along with fundamental object-oriented principles such as SOLID and Dry principles.

Used Design Patterns

Factory Pattern (CardFactory Class & Animal Enum):

The CardFactory class encapsulates the creation of different types of cards used in the game, such as ChitCard and labels for volcano cards. This class abstracts the instantiation process, making the system independent of how the products are created, composed, and represented.

The introduction of the Animal enum has further refined the abstraction process by standardizing the image references used in card creation, enhancing maintainability and type safety.

Pros: By centralizing card creation, this pattern allows for more manageable code, making it easier to introduce new card types or change existing ones without affecting the classes that use them.

Singleton Pattern (ChitCardFlipManager Class):

The ChitCardFlipManager ensures that there is only one instance managing the state of flipped cards across the application. This class controls the flipping logic, ensuring that no two cards can be flipped simultaneously.

Benefits: Singleton usage here is crucial for maintaining consistent and error-free game state transitions, particularly in the flipping mechanism of cards, which is central to gameplay.

These patterns are supported by robust object-oriented design:

SOLID Principles:

Single Responsibility Principle (SRP):

The BoardPanel class handles the GUI representation of the game board, focusing on layout and painting (paintComponent method). It also loads images (loadImages method), which can be considered a separate responsibility but still related to the board's appearance.

CardFactory is responsible for creating different types of cards (createChitCard, createVolcanoCard, createPlayerIndicator). Each factory method handles a specific type of card creation, adhering to SRP.

ChitCard manages the state of an interactive game card and handles flipping logic (isFlipped, setFlipped, unflip). It also paints the card based on its state (paintComponent method).

Open/Closed Principle (OCP):

Animal enum allows for the easy addition of new animal types without altering existing card creation logic, directly supporting the OCP. The remaining code doesn't explicitly demonstrate adherence to the OCP. However, by using factory methods like createChitCard, createVolcanoCard, etc., it allows for extension (e.g., adding new card types) without modifying existing code, indirectly supporting the principle.

Dependency Inversion Principle (DIP):
The code adheres to DIP by relying on interfaces (e.g., JPanel, JLabel) instead of concrete implementations, promoting flexibility and easier testing.

DRY Principle:
Code Reuse: The code reuses common functionalities such as card creation (CardFactory) and flipping management (ChitCardFlipManager) across different parts of the application, promoting code reuse and avoiding duplication.
Modularity: The design is modular, with clear interfaces between components like UI elements and game logic, facilitating independent development and testing of each module.
Encapsulation: Each class has well-defined responsibilities, encapsulating specific behaviours and data necessary for its function, such as BoardPanel managing all rendering-related tasks.

1.2 Alternative Designs
The Alternative design options included a more procedural approach and a less modular class structure:

Procedural Approach
Integrating game logic directly within UI classes, such as handling card flipping logic directly in the Card or BoardPanel classes.
Cons: This approach would lead to high coupling and low cohesion, making the system less flexible and harder to maintain or extend. Changes in game logic could necessitate significant modifications to the UI code, violating the principle of separation of concerns.

Monolithic Class Design
Using fewer, larger classes to handle multiple aspects of the game logic and UI rendering.
Cons: Such a design would make it difficult to manage complexity as the game evolves. Testing would become cumbersome, and the ability to reuse components in different parts of the game or different projects would be severely limited.

The chosen design, utilizing both the Factory and Singleton patterns supported by strong object-oriented principles, offers a scalable and maintainable framework. This design not only facilitates easier updates and potential expansions of the game but also ensures that the codebase remains manageable and robust against future requirements or changes. The alternatives, while simpler initially, would not provide the flexibility and maintainability required for a dynamic game like "Fiery Dragons".

1.3 Design Rationales (explaining the 'Whys')

1.3.1 Key Classes

BoardPanel
Handles the graphical representation of the game board. It loads and displays the dragon image at the centre, draws the board background, and manages the layout.

Rationale: The BoardPanel class is central as it serves as the visual foundation upon which other game elements are layered. Separating the board's graphical management into its own class adheres to the Single Responsibility Principle, ensuring that the class only handles rendering and not game logic or state management.

CardFactory
Implements the factory pattern to create different types of card-related UI components like ChitCard and labels for volcano cards.

Rationale: Using a factory pattern here allows for encapsulation of the creation logic of cards and related components. It simplifies maintenance and scalability by centralizing the creation logic, making it easier to update the game's visual elements or add new card types without altering the consumer classes.

Animal <<Enum>>
Introduces an enumeration for the different types of animals represented in the volcano cards, each associated with a specific image file.

Rationale: The Animal enum serves to encapsulate the properties of different animal types in a single, centralized location. This promotes the Open/Closed Principle as new animal types can be added without altering existing code elsewhere in the system. Using an enum ensures that the animal types are constants and prevents issues like duplication or inconsistency across the application.

1.3.2 Key Relationships

Composition between GamePanel and Card
Explanation: GamePanel contains a list of Card objects representing different cards on the board.
Rationale: This is a composition relationship because Card objects do not logically exist independent of GamePanel. When GamePanel is destroyed or reset, the Card objects it manages should also cease to exist, reflecting their tightly coupled lifecycle.

Aggregation between GameFrame and BoardPanel
Explanation: GameFrame includes BoardPanel but does not strictly own it.
Rationale: Aggregation is used here to represent the relationship where BoardPanel can exist independently of GameFrame. This flexibility allows for the potential reuse of BoardPanel in different contexts or frames within the application, supporting modularity.

Use of Enums in CardFactory

Explanation: The CardFactory utilizes the Animal enum to determine the appropriate image for volcano cards, streamlining the creation process based on predefined properties.

Rationale: This approach leverages the Single Responsibility Principle by delegating the responsibility of managing animal-related properties to the Animal enum, allowing the CardFactory to remain focused on component creation. It also exemplifies the Dependency Inversion Principle by depending on the abstraction provided by the enum rather than concrete implementations, which enhances flexibility and maintainability.

1.3.3 Inheritance Decisions

The project utilizes inheritance minimally, focusing instead on composition and interface usage where practical. For instance, UI components like BoardPanel and ControlPanel extend JPanel to inherit basic panel capabilities while implementing specific functionalities.

Justification: This decision avoids the rigidity and increased complexity associated with deep inheritance hierarchies. By favouring composition, the design remains flexible, making it easier to modify or extend component behaviours without risking side effects from base class changes.

MONASH University

1.3.4 Cardinality Decisions

0..1 Cardinality for ChitCard flipped state

A ChitCard can be in a flipped state (showing its face) or not.

Rationale: This cardinality allows for the optional flipped state of each card, supporting game dynamics where the visibility of a card's face is a temporary and reversible condition.

*1.. Cardinality between GamePanel and Card**

Each GamePanel must manage at least one Card, but potentially many.

Rationale: The game requires at least one card to function but typically involves multiple cards. This cardinality ensures that GamePanel always has cards to manage, enforcing a basic precondition for the game's operation.

Video Demo Link

https://youtu.be/iM3xT_F8iwY

In this video, we talk about how to create a .JAR file and how to launch the game. We also discuss the game functionalities implemented