REQ 1: Divine Beast Dancing Lion

Task Description

The objective of this task is to add the **Divine Beast Dancing Lion** character with unique, rotating "divine powers" (Wind, Frost, and Lightning) and its initial basic attack(biting). Each power grants the lion a special attack with specific effects:

- **Wind Power**: Moves the target to a new random location adjacent to the divine beast.
- **Frost Power**: Causes the target to slip and drop all items (if on water).
- **Lightning Power**: Damages nearby actors with an extra effect if on water.

Also the divine powers can be switched and the probability of that happening is 25%.

The design should support adding new divine powers in the future and allow the lion to switch powers dynamically. This functionality must adhere to SOLID principles.

Design 1: Using an Abstract Class `DivinePower` with Subclasses

Design Outline

1. **`DivinePower` (Abstract Class)**: It is an abstract class which represents the general concept of a divine power and contains abstract methods for the special attack behaviours and power switching. It has a `specialAttack` method which is for special attacks and switchPower method which is for switching the powers based on the requirement.
2. **Concrete Power Classes**: We have different power classes for different powers such as `WindPower`, `FrostPower`, and `LightningPower`, `and each of them` extend `DivinePower` Class and implement their unique `specialAttack` and switchPower methods.
3. **`DivineBeastDancingLion` (Concrete Class)**: Manages the main behaviour of the beast, holds a `DivinePower` reference for the current power, and uses it to execute attacks and switch between powers. It uses Enemy Class as a base and takes the help of how the FurnaceGolem was implemented using Enemy, WanderBehaviour, FollowBehaviour, and AttackBehaviour Class. It also implements a gate function, using Gate Class when the Divine Beast is defeated, and in its last position a gate emerges from which Tarnished can go to the Tower Settlement

Pros and Cons

Pros	Cons
Extensibility: Adding new divine powers (e.g., FirePower) will only require creating a new subclass of `DivinePower`, aligning with **OCP**(Open Closed Principle).	**Complexity**: It requires multiple classes (one for each power), which may feel cumbersome for small-scale projects.
Single Responsibility: Each of the `DivinePower` subclass handles only its specific power's behaviour, aligning with **SRP** (Single Responsibility Principle).	**Potential Overhead**: Switching powers involves instantiating different classes, which may introduce minor performance overhead.
Polymorphism: `DivineBeastDancingLion` interacts with `DivinePower` polymorphically, making it flexible to work with different power types, satisfying **LSP**.	**Minor Redundancy**: Each power subclass has its own `specialAttack` method, which could be seen as repetitive if they only differ slightly.
Code Reuse: Common logic for powers can be placed in `DivinePower`, following **DRY** principles. Also if we extend our code by adding other power classes, we could just add that power properties by using the DivinePower abstract class.	**Dependency**: `DivineBeastDancingLion` depends on `DivinePower` for its attacks, which could become an issue if power behaviour varies significantly.

Class Structure and Interactions

1. **`DivinePower` (Abstract Class)**
    - `specialAttack(Actor lion, Actor target, GameMap map)`: Abstract method for the power's unique attack.
    - `switchPower()`: Determines and returns the next power.
2. **Concrete Divine Powers (`WindPower`, `FrostPower`, `LightningPower`)**
    - Implement `specialAttack` based on the power's unique behaviour.
    - Each subclass returns the appropriate next power type in `switchPower` `based on the requirement provided`.
3. **`DivineBeastDancingLion`**
    - Holds a reference to a `DivinePower` instance.
    - Executes usual bite and specialPower attacks and calls `switchPower` to change powers based on probabilities.

Design 2: Managing Powers Directly within `DivineBeastDancingLion`

Design Outline

1. **No Abstract Class for Powers**: Divine powers are methods within `DivineBeastDancingLion` (e.g., `windAttack`, `frostAttack`, `lightningAttack`).
2. **Power Switching Logic**: Managed directly in `DivineBeastDancingLion` with a switch statement or probability-based logic for determining the next power.
3. **Single Class**: `DivineBeastDancingLion` is responsible for all power behaviours without relying on subclasses.

Pros and Cons

Pros	Cons
Simpler Structure: Fewer classes will be involved in this structure, which could simplify understanding and reduce initial setup time.	**Violation of Single Responsibility Principle**: `DivineBeastDancingLion` now manages both its own behaviour and each power's behaviour, making it a "God" class with multiple responsibilities.
No Dependency on External Power Classes: Everything is contained within `DivineBeastDancingLion`, reducing external dependencies.	**Difficult to Extend**: Adding new powers requires modifying `DivineBeastDancingLion`, violating **OCP** and making future modifications more error-prone.
Centralized Logic: Attack and power-switching logic are concentrated in one place, potentially which simplifies debugging.	**No Polymorphism**: Without power classes, the design loses the ability to treat each power polymorphically, limiting flexibility and violating **LSP**.
Reduced File Count: No extra classes mean fewer files, which could be easier to manage in a small project.	**Repetitive Code**: Duplicate code for power-specific behaviours can arise if each power's behaviour is not generalized, violating **DRY**.

Class Structure and Interactions

1. `DivineBeastDancingLion`
    - Methods like `WindAttack`, `FrostAttack`, and `LightningAttack` define each power's unique behaviour.
    - A main method controls power switching directly, e.g., using a switch statement to manage different attack behaviours.

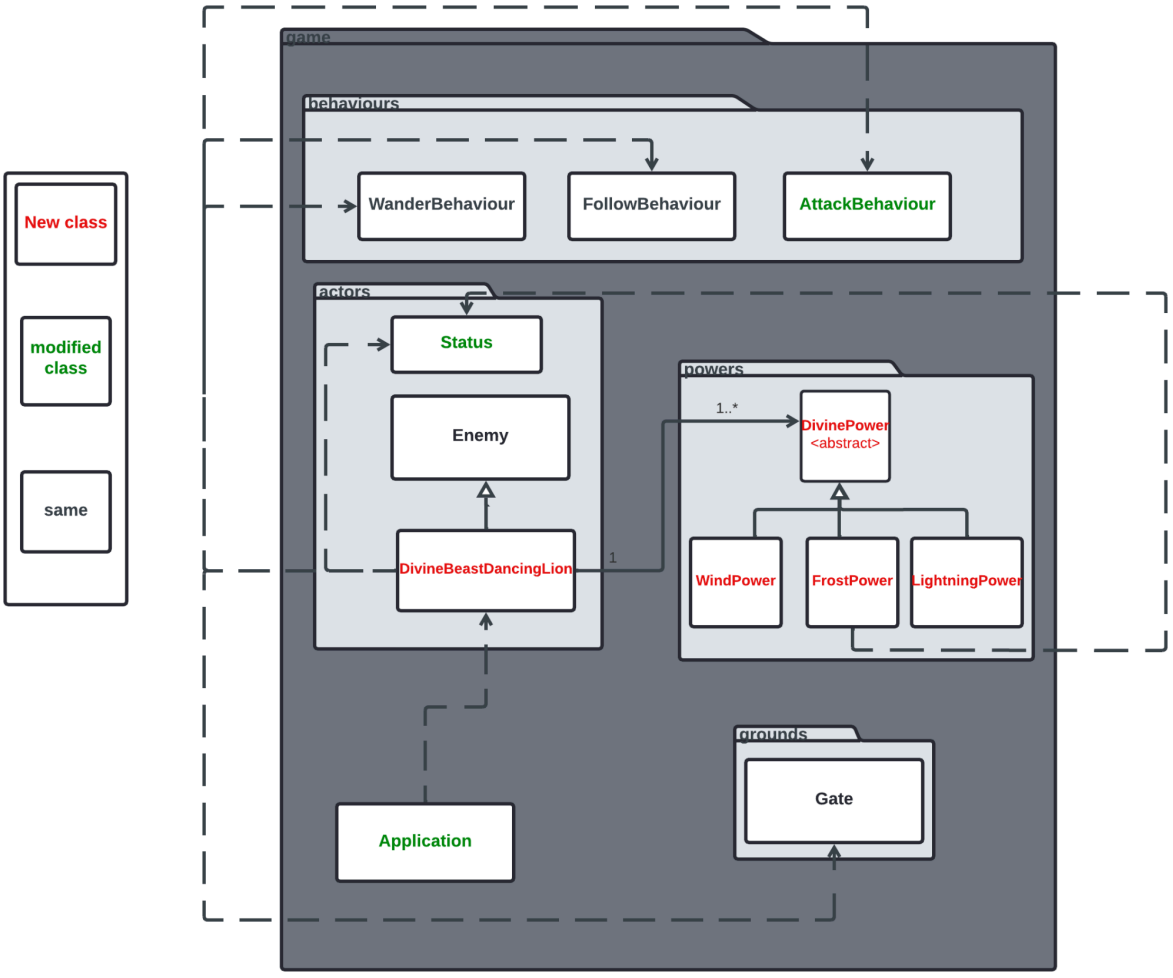
Design Recommendation

In conclusion, **Design 1** is recommended because it adheres to the core OOP principles, making it more scalable in future, modular, and flexible. The use of abstraction of DivinePower reduces duplication chances, and increases extensibility with minimum changes. This design provides a more solid and simpler framework for further additions and it is easier to edit this design in comparison to **Design 2**.

In contrast, **Design 2** offers a simpler structure with fewer classes, it leads to high coupling and increased responsibilities in `DivineBeastDancingLion`, making it harder to extend and maintain. It also violates core OOP principles.

Therefore, **Design 1** is better suited for scalability and maintainability, even if it involves a few more classes upfront.

Shown Below is the UML diagram of this design.



Changes made for Assignment 2 code

- **Flask of Healing behaviour Correction**: Previously, the `FlaskOfHealing` item increased both the `maxHealthPoints` and `currentHitPoints` when consumed. I updated it to only restore up to the actor's `currentHitPoints` limit, without exceeding the maximum health. This change ensures that healing remains within logical boundaries, enhancing game balance and preventing unintended health inflation.
- **Null Pointer Exception Prevention**: I identified and resolved several `NullPointerException` issues that occasionally disrupted gameplay. These changes improve code stability and prevent unexpected crashes in various scenarios, especially when certain attributes or items were not properly initialized.
- **AttackBehaviour and Furnace Golem Refactoring**: I refactored the `AttackBehaviour` and `FurnaceGolem` classes to improve modularity and reusability. These changes allow `AttackBehaviour` to be extended to other classes, such as `DivineBeastDancingLion` or any other `Enemy` class, making the code more flexible and easier to maintain or extend for future enemies with similar attack patterns. This design supports the Open/Closed Principle (OCP) by enabling behaviour reuse without modifying existing code.