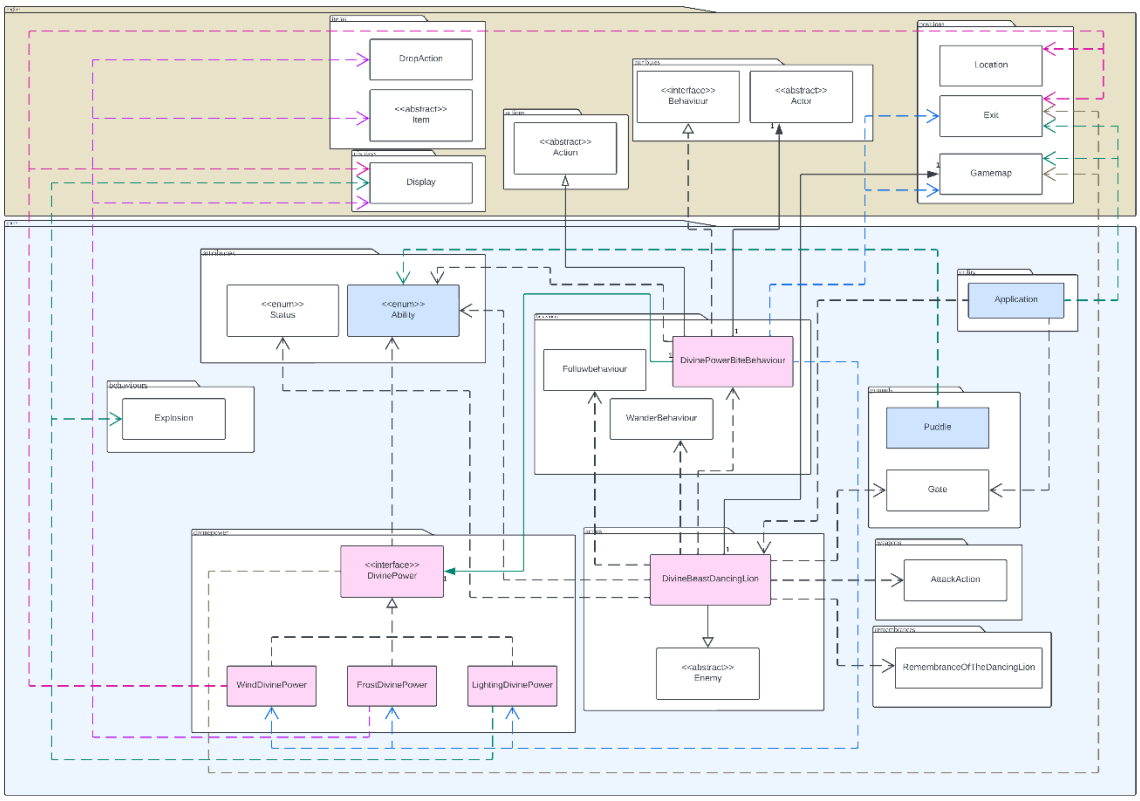
REQ1: Divine Beast Dancing Lion



Pink = Class Created

Blue = Class Modified

REQ 1 UML Diagram

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr><td>Modified Application class</td><td>Role & Responsibility:

Handle the addition of a new map, Stagefront, a new enemy, Divine Beast Dancing Lion, and a new gate, Belurat Tower to Stagefront Gate.

Integrate the new game elements, ensuring that the player encounters the boss fight after passing through the gate to the Stagefront map.

Relationship:
Dependency with newly created enemy, gate, game map, and exit components</td><td rowspan="2">Alternative Solution: Concrete Divine Beast Dancing Lion with all implementation in one class

Pros and Cons</br>
<table><tr><th>Pros</th><th>Cons</th></tr><tr><td>Simplified design: The entire logic, behavior, and abilities of the Divine Beast Dancing Lion would be encapsulated in a single class, making it easier to understand in the short term.</td><td>Violation of SOLID principles: By cramming all responsibilities into a single class, this solution violates the Single Responsibility Principle (SRP). Instead of separating concerns, one class would manage all behavior, divine powers, and actions, making future modifications harder.</td></tr><tr><td>Quick implementation: Having all functionality in one place would reduce the time needed to code the solution initially.</td><td>Code smell: The design would create a "God Class" problem, where one class is too large and complicated. This reduces maintainability and increases the risk of introducing bugs, as any small change could affect multiple areas.</td></tr><tr><td></td><td>Difficult to extend: Adding new enemies, abilities, or behaviors in the future would require significant changes to this</td></tr></table></td></tr>
<tr><td>Extended Ability class with new abilities:
-Divine Power
-Wind Devine Power
-Frost Devine Power
-Lightning Devine Power
-Electric Weakness
-Slippery</td><td>Role & Responsibility:

enrich the game's mechanics, allowing specific actions and behaviors to be triggered based on the abilities possessed by the actor.

Relationship:

While there are no direct dependencies or associations with other classes, these abilities are integral to the new behaviors and ground for the boss fight.</td></tr>
</table>

			large class, breaking the Open/Closed Principle (OCP) and potentially introducing code duplication.
Created Divine Beast Dancing Lion (extends enemy)	**Role & Responsibility:** The *DivineBeastDancingLion* class, which extends from the *Enemy* superclass, represents a boss-level enemy with significant hit points (10,000 HP). Once the player crosses the gate to the *Stagefront*, the boss fight begins, preventing the player from returning to Belurat until the boss is defeated. This creates a clear gameplay progression. The boss is designed to engage in regular attacks, dealing 150 damage with a 30% hit chance, and has specific behaviors such as following the player until either one dies. Upon defeat, it drops a "Remembrance of the Dancing Lion" and opens a gate, allowing the player to return to Belurat. **Relationship:** - Extends enemy - Associated to gamemap - Dependency to AttackAction, Gate, Remembrance of the Dancing Lion, Divine		

Final Solution: Modular and SOLID Compliant Design

Design Overview

In the finalized solution, I opted for a design that adheres to SOLID principles and avoids code smells by distributing responsibilities across well-defined classes and interfaces. Here's a summary of the modified and newly created components:

- **Modified Application Class:** Responsible for handling the addition of the new map, enemy (Divine Beast Dancing Lion), and gate. It integrates the new game elements to ensure smooth gameplay flow, such as triggering the boss fight upon crossing the gate.
- **Extended Ability Class:** Introduces new abilities like Wind Divine Power, Frost Divine Power, and Lightning Divine Power. These abilities enrich the gameplay by providing unique effects based on the player's and enemy's interactions with certain ground types.
- **Created Divine Beast Dancing Lion Class (Extends Enemy):** This class is dedicated to handling the Divine Beast's attributes (e.g., 10,000 HP) and interaction logic (e.g., attacking the player, triggering behaviors).

<table>
<tr><td></td><td>Power Bite Behaviour, Follow behaviour, Wander behaviour, Ability, Status</td><td rowspan="2">The class follows the SRP and delegates behavior to specific behavior classes.

• Created Divine Power Bite Behavior (Extends Action, Implements Behavior): This class manages the Divine Beast's ability to switch between divine powers during combat. It abstracts special attacks, ensuring that the code for divine powers is isolated, reducing duplication and complexity.

• Created Divine Power Interface: The interface enforces a consistent contract for each divine power (Wind, Frost, Lightning). Each power is implemented in its own class, following the Interface Segregation Principle (ISP) and reducing unnecessary dependencies.

• Added Abilities to Puddle Class (SLIPPERY, ELECTRIC_WEAKNESS): These abilities connect the environment with the divine powers, making the interaction between the player, puddles, and divine powers more dynamic and tactical.

Reasons of Decision:

SOLID Principles
1. Single Responsibility Principle (SRP)
Each class in this design has a clearly defined responsibility:
• Enemy Class (DivineBeastDancingLion): Focuses on enemy-specific logic, such as hit points, movement, and behavior during the boss fight.
• Divine Powers (WindDivinePower, FrostDivinePower, LightningDivinePower):</td></tr>
<tr><td>Created Divine Power Bite Behaviour (extends Action implements Behaviour)</td><td>Role & Responsibility:

Encapsulates the special divine powers possessed by the DivineBeastDancingLion. Each power—Wind, Frost, and Lightning—has its unique mechanics triggered with a 25% chance to change between divine powers during an attack. This behavior adds an element of unpredictability to the boss fight. Additionally, the divine powers' special attacks are independent of the normal attack, enhancing the tactical complexity.

Relationship:

- Extends Action and implements Behaviour.
- Association with Actor, DivinePower
- Dependency with Ability, Exit, Gamemap, winddivinepower, frostdivinepower, lightningdivinepower</td></tr>
</table>

<table>
<tr><td></td><td></td><td rowspan="3">Each power encapsulates its special mechanics, adhering to the DivinePower interface and separating the logic for wind, frost, and lightning attacks.
• Behaviors (DivinePowerBiteBehaviour): Encapsulates the selection of divine powers and attack behavior, thus separating behavior concerns from the DivineBeastDancingLion class.
• Abilities (SLIPPER, ELECTRIC_WEAKNESS): Enrich the game's mechanics, allowing specific actions to be triggered based on an actor's context in the environment.
This clear separation of responsibilities ensures that future changes or extensions can be easily accommodated. For example, new enemies or abilities can be added without modifying the existing code.
2. Open/Closed Principle (OCP)
The design adheres to the OCP because new enemies, powers, or behaviors can be added without altering existing classes. The introduction of DivinePower as an interface ensures that new divine powers can be added in the future (e.g., FireDivinePower) without altering the core behavior of the DivineBeastDancingLion class. The Ability system also supports future expansion by allowing new environmental effects to be incorporated without modifying existing ground logic.
3. Liskov Substitution Principle (LSP)
The DivineBeastDancingLion and its divine powers are designed such that any new enemy or power can substitute the existing ones without breaking the system. For instance, any subclass of DivineBeastDancingLion can be treated as an enemy,</td></tr>
<tr><td>Created Divine Power Interface</td><td>Role & Responsibility:

Ensuring that each power adheres to a consistent contract for performing special attacks and transitioning between powers.

Relationship:

- Dependency with Actor, Game map, Ability</td></tr>
<tr><td>Created

Wind Divine Power, Frost Divine Power, Lighting Divine Power classes

(implements Divine Power)</td><td>Role & Responsibility:

Each divine power (Wind, Frost, Lightning) is implemented as a separate class that follows the DivinePower interface with different logic, wind blow the target to a random location around the actor, frost drop all portable item of target if target standing on ground that consist the ability slippery, lightning creates a shockwave that shock target and double up the damage if target standing on ground that consist the ability electric_weakness.

Relationship:</td></tr>
</table>

<table>
<tr><td></td><td>- Implements Divine Power
- Wind: Dependency to Display, Location and Exit
- Frost: Dependency to Drop Action, Item and Display
- Lightning: Dependency to Explosion and Display</td><td rowspan="2">and any new class implementing DivinePower can perform special attacks, ensuring compatibility with existing behaviors and interactions.
4. Interface Segregation Principle (ISP)
The DivinePower interface allows different powers to implement their unique logic. By segregating the responsibilities of each power, we prevent the need for unnecessary methods in each power class. This clean separation makes it easier to manage different powers independently without coupling them with each other.
5. Dependency Inversion Principle (DIP)
The system relies on abstractions such as DivinePower and Behaviour rather than concrete implementations. The DivineBeastDancingLion class interacts with behaviors and powers through their abstractions, allowing flexibility and extensibility. Future powers and behaviors can be introduced without modifying the core enemy logic.

DRY Principle
The design eliminates code duplication through shared behavior logic and reusable interfaces:
• The DivinePower interface avoids duplicating logic for each power’s special attack.
• The DivinePowerBiteBehaviour centralizes the logic for switching between different divine powers, ensuring that the same behavior can be applied to multiple enemies in the future.
• The use of Ability in the Puddle class provides a shared mechanism for handling ground-based interactions without repeating conditional logic across multiple power classes.</td></tr>
<tr><td>Added new Ability to Puddle class</td><td>Role & Responsibility:

The Puddle class was modified to include new capabilities—SLIPPERY and ELECTRIC_WEAKNESS—which tie into the effects of the divine powers. These modifications allow the Frost power to cause the player to slip and drop their inventory if standing on a puddle, while the Lightning power doubles its damage when the player is standing on water.

Relationship:
- Dependency to Ability</td></tr>
</table>

		Connascence Connascence refers to the degree of interdependence between modules. In this design, the connascence has been minimized to reduce the fragility of the codebase. **1. Connascence of Name** The design uses clear, self-explanatory names for abilities (e.g., SLIPPERY, ELECTRIC_WEAKNESS), powers (e.g., WindDivinePower), and behaviors (e.g., DivinePowerBiteBehaviour). This makes it easier to understand and work with the code without introducing naming conflicts or misunderstandings. **2. Connascence of Type** The DivinePower interface defines a contract for divine powers, ensuring that all powers share the same type and structure. By depending on this abstraction, the system avoids tightly coupling the logic of DivineBeastDancingLion to specific power implementations, allowing flexibility for future expansions. **3. Connascence of Meaning** There is some level of connascence between the ground type (e.g., puddles with ELECTRIC_WEAKNESS) and the powers (e.g., LightningDivinePower), but this is intentional as these interactions are part of the game's mechanics. The system can easily adapt to new mechanics by adding more ground types or abilities.

		Code Smells and Refactoring This design aims to avoid common code smells: **1. Large Class** By separating behaviors and divine powers into distinct classes, we avoid the "large class" smell that could occur if all the logic were centralized in the DivineBeastDancingLion class. This results in cleaner, more manageable code. **2. Duplicated Code** The use of interfaces and shared logic (such as the Ability and DivinePower systems) prevents code duplication. Any additional enemies or powers can reuse the same behaviors, reducing the risk of repeating similar logic across different classes. **3. Feature Envy** Classes are designed to encapsulate their own functionality. For example, DivinePower encapsulates the power-specific logic, while the DivineBeastDancingLion focuses on enemy-specific behavior. This prevents "feature envy," where one class would depend too heavily on another class's internals. **Pros and Cons of the Design** **Pros:** • **Flexibility**: The design is highly flexible and open to extension. New powers, behaviors, or enemies can be introduced without altering existing code. • **Separation of Concerns**: Each class has a clear responsibility, making the system easier to understand, maintain, and extend. • **Adherence to SOLID Principles**: The design fully adheres to SOLID, ensuring clean and maintainable code.

		• **Code Reuse**: Shared logic, such as behaviors and abilities, reduces redundancy and prevents duplication across the system. **Cons:** • **Increased Complexity**: The separation of responsibilities into multiple classes can make the system more complex to understand initially. • **Performance Overhead**: The use of multiple behaviors, abilities, and checks might introduce some performance overhead, though this is unlikely to be significant. **Future Extensibility** **1. New Types of Divine Powers** There may be future divine powers beyond Wind, Frost, and Lightning. To accommodate this, the design follows the Open/Closed Principle (OCP) by abstracting divine powers through the DivinePower interface. This means that adding new powers, such as FireDivinePower or WaterDivinePower, can be done without altering the existing DivineBeastDancingLion class or other behaviors that interact with divine powers. Each new power can implement its unique logic, while still adhering to the contract defined by the DivinePower interface. **2. Other Entities Changing "States"** In the future, entities other than actors, such as environmental objects or certain items, may also have the ability to change their "state" and trigger interactions similar to divine powers. The system is designed to support this extension by generalizing state transitions and behaviors through the use of the **DivinePower** interface and **Ability** system.

		3. Flexible Starting Divine Power Currently, the **Divine Beast Dancing Lion** starts with the **WindDivinePower** by default. However, the design can easily accommodate starting with other powers, such as Frost or Lightning, without hardcoding this logic into the class. By abstracting the state transitions and making the initial state configurable, future changes to the starting divine power can be handled dynamically at runtime. **5. Support for Multiple Instances of Divine Beast Dancing Lion** In future expansions, multiple instances of the **Divine Beast Dancing Lion** may appear with different behaviors or divine power sets. The current design ensures that each instance can have its unique set of powers and behaviors by allowing the **DivinePowerBiteBehaviour** to be configured separately for each instance. This allows different **Divine Beast Dancing Lions** to behave differently, while still leveraging the same shared logic. **6. Compatibility with Other Enemies or Bosses** Other future bosses or enemies could also inherit or interact with the divine power system. The system is designed to allow any future boss class to use the existing **DivinePower** and **Behavior** components. By relying on abstractions, the system remains flexible for extending divine powers to new enemies without duplicating code.