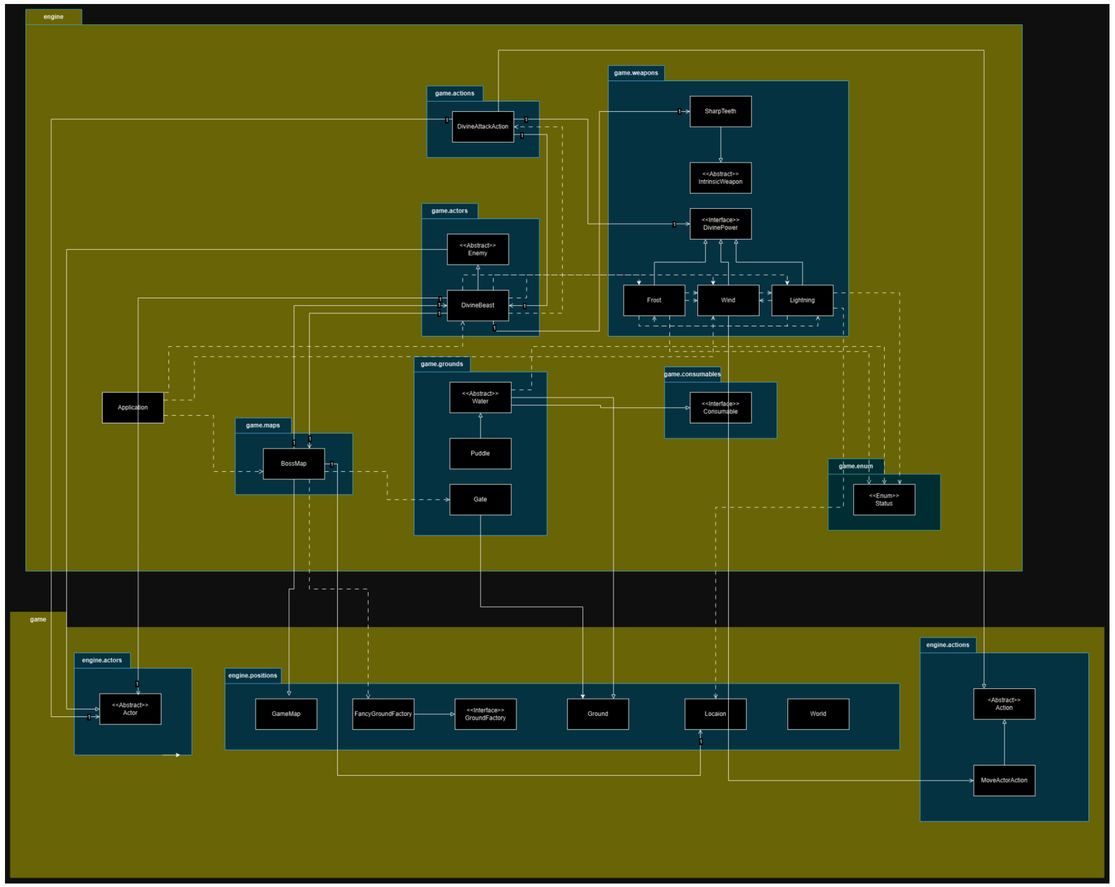
Assignment 3 Requirement 1 Design Rationale



Classes Modified/Created	Roles/Responsibilities	Rationale

Assignment 3 Requirement 1 Design Rationale

BossMap DivineBeast	Manages specialised map environment for boss encounters, including boss and player spawn points, and the return gate creation upon the boss's defeat. Represents enemy in the game.	**Solution:** A specific BossMap class was created that extended from GameMap, and represented a specialised GameMap for boss interactions and fights. It contained a method createReturnGate(), which sets the return location of the Gate out of the map, upon the boss's defeat. This is achieved by accepting a returnLocation as a Location instance within the parameter when a BossMap instance is instantiated. The createReturnGate() method instantiates a gate as an instance of the Gate Ground class. This class also handles the logic of beast and fighter spawn points, using the tick() method to check if an actor arrives at the spawn point from a gate from another map instantiating an instance of the DivineBeast at the boss's spawn point if there is an actor. The DivineBeast was implemented to extend from the Enemy abstract class, as it shared similar intended outcomes and methods, including the allowableactions() method which checked and allowed for behaviours such as FOLLOW. While the creation of the return gate is left to the BossMap class, the unconscious() method of defined and overridden in the Actor class **Reasoning:** 1. By isolating the responsibility of the logic behind return creation to the BossMap class, the DivineBeast maintains a focused role on representing an Enemy/Boss on a map and the associate fight logic. (**Single Responsibility Rule**). While the createReturnGate method is called within the DivineBeast class, it is called on the map associated with the DivineBeast and still does not have to handle any of the actual implementation logic. It seemed logical to create the gate in the DivneBeast class as it already had an unconscious() method that is automatically called when the actors reaches 0. This implementation avoids redundant code and ensures code maintainability 2. By placing the location for the return gate as a parameter for the BossMap, we maintain a clear separation of of concerns, where each map manages its own environment and transitions. I.e. A developer can add or modify boss maps with different return locations without altering the core logic of DivineBeast or BossMap.(**Open/Closed Principle)** 3. Extension from GameMap when creating the BossMap was logical as it adheres to **Liskov Substition Principle**. This means that it can be referenced as GameMap in higher level code without requiring specialised handling for this new class. For example in the application class,

when world.addGameMap() is called to add a new map to the world, any instance of a BossMapcan be passed as a parameter as it is recognised as a GameMap. This promotes code consistency and reusability, as duplicate methods do not need to be created.

4. Similarly, this improves code reusability and maintainability as high-level modules such as World class don’t depend on low-level classes such as BossMap or DivineBeast and are therefore not affected by their specific changes. Instead it depends on the Actor and Action classes and this abstraction allows for decoupling between high-level and low-level modules. I.e. BossMap provides a structured way to manage boss-specific locations and spawn points, ensuring all boss events are isolated from the rest of the GameMap logic. (**Dependency Inversion Principle**).

5. Common methods such as .at() are implemented within the GameMap class and inherited from the BossMap class so that is left only to handle boss-specific functionality like creating return gates, avoiding redundant code. (**DRY**)

Limitations:

1. Manual Configuration: The current implementation requires users to manually input the return location as a parameter. This creates the potential for human error, such as invalid coordinates.

2. The return gate creation strictly depends on the unconscious() method in DivineBeast, limiting flexibility if alternative ways to end the boss fight are introduced.

   To expand the gate creation logic to a broader context, a possible future improvement could be creating a BossEncounterGate class that handles only gate-related configurations for boss encounters, as well as a BossMapManager, which coordinations boss setup and management,. BossMapManager would encapsulate the logic for handling multiple boss maps, all with different bosses and return gate locations, as each BossEncounter Gate instance would be injected with a list of returnlocations. This ensures improved adherence to SRP and OCP, as no boss concrete class e.g. DivineBeast is responsible for calling a method to create the return gate.

   It is important to note, however that this would also increase overhead which could lead to possible confusion between developers, as the number of different classes increases. Especially, when the unconscious() function already exists for all actors and therefore bosses and is called, it seems logical to call the method to create the return class in this function.

Assignment 3 Requirement 1 Design Rationale

		Alternative Solution: The DivineBeast directly creates the return gate on its defeat, but hardcoding the returnlocation into the unconscious() method. This would centralize both Divine Beast representation and return gate creation mechanics within the one class **Limitations of Alternative Solution:** 1. Violation of **SRP** → The DivineBeast class is responsible for handling both beast representation/combat logic, as well as map transition logic. 2. The unconscious() method would always teteleport the actor to the only one destination, making this logic rigid and inflexible. This doesn't allow for actor input or dynamic destination selection. To extend you have to modify existing code, which could break other parts of the problem (violates **OCP**)

Classes Modified/Created	**Roles/Responsibilities**	**Rationale**
Wind Frost Lightning DivineAttackAction	.Represent individual divine powers, applying specific effects on a target and providing the logic to switch powers. Handles the execution of a	**Solution:** A DivinePower interface was created to standardise the behaviour of different divine powers, ensuring that each subclass that extends from it (E.g. Wind, Frost and Lightning) adheres to a consistent structure, while allowing them to implement the unique effects in each concrete class. Each DivineBeast instance has a DivinePower attribute, and methods such as getNextPower() and getAttackDescription(), allow the DivineBeast to change to a different power before every attack (if 25% chance requirement is met) and display on the menu the outcome of each divine power attack, respectfully. It was logicial to make DivinePower an interface rather than an abstract class, as the subclasses had no shared attributes. The DivineAttackAction class takes in a DivinePower and a DivineBeast as its parameter, as well as its target, and executes the current divine power's divineAttack on the target. It also calls attackDescription()

Assignment 3 Requirement 1 Design Rationale

	Divine Beast's divine power.	to provide a summary of the attack's impact. Therefore for every tick() of the game, when playTurn is called in the divineBeast class, while a dancePartner exists a new DivineAction will be returned and executed in the proccessActorTurn() method in the World class. **Reasoning:** 6. By isolating the responsibility of the logic behind the effect of each Divine Power to the concrete classes, this ensures that each class that implements the Divine Power interface is only responsible for their own logic. (**Single Responsibility Rule**). Similarly, the isolation of the Divine Attack logic into the DivineAttackAction, ensures that the DivineBeast class is not responsible for executing the attack or broadcasting the outcome. It is only responsible for representing a DivineBeast and determining eligible actions for the Beast each turn. (**SRP**) This ensures code maintainability, making it easier to debug. 7. Implementation from the DivinePower interface, supports modularity and encourages extensibility as additional powers can be introduced and their attack logic can be implemented without altering the code of existing divine powers*.(**Open/Closed Principle)** Moreover, abstraction from the Action class through the creation of DivineAttackAction, ensures that special features (e.g. damage boosts) can be added for new bosses by extending from DivineAttack without having to modify DivineBeast class (**OCP**) 8. Extension from the Action abstract class when implementing DivineAttackAction was logical as it adheres to **Liskov Substitution Principle**. This means that it can be referenced as an Action in higher level code without requiring specialised handling for this new class. E.g. in the proccessActorTurn() method in the World class, when action.execute() is called, where action is recognised as an Action class instance, if a DivineAttackAction is returned from actor.playTurn() it will be recognised. This promotes code consistency and reusability, as duplicate methods do not need to be created.(**DRY**) 9. Similarly, this improves code reusability and maintainability as high-level modules such as World class don't depend on low-level classes such as Frost or Wind and are therefore not affected by

Assignment 3 Requirement 1 Design Rationale

their specific changes. Instead it depends on the Action classes, as DivineAttackAction handles the power execution, and this abstraction allows for decoupling between high-level and low-level modules. (**Dependency Inversion Principle**).

Similarly, abstraction from the Divine Power interface ensures that when DivinePower is called on in DivineBeast it references the interface, not the low-level concrete classes and therefore will not be affected by their specific changes, or additional power classes (directly of course, it would mean they have another type of Divine Power to use).

Limitations:

3. *A limitation of this implementation is that each power directly references the next power in sequence, meaning changes to the sequence including the addition of new divine powers or changes in the likelihood of power switching involves altering existing hard code, therefore violating **OCP**.

   For future improvement, it would be wise to implement a centralised power management system, namely a DivinePowerFactory, responsible for creating different types of powers, and a PowerManager, which dictates which power the DivineBeast should use next. Not only does this improve adherence to **SRP**, but more importantly **OCP**, as each DivineBeast has its own DivinePowerManager attribute and there new or changed powers can be managed in said classes rather than DivineBeast itself.

4. The abstraction from Action to create a DivineAttackAction to create an intermediate for divine power execution so it is not left to the DivineBeast class to be handled, increases overhead. The complexity level of abstraction may outweigh the improved SRP compliance, and it may cause confusion between developers as the number of classes increases.

Alternative Solution:
Managing divine powers within DivineBeast. I.e. executing the effect of the current Divine Power in the PlayTurn() method of the Divine Beast

Limitations of Alternative Solution:

Assignment 3 Requirement 1 Design Rationale

		3. Violation of **SRP** → The DivineBeast class is responsible for handling both beast representation/combat logic, as well as Divine Power execution. 4. This also means that unique behaviours through additional methods in the DivinePower subclasses could not be handled, making this logic rigid and inflexible. To extend you have to modify existing code, which could break other parts of the problem (violates **OCP**)

Classes Modified/Created	**Roles/Responsibilities**	**Rationale**
Frost Status Water	.Represent individual divine powers, that causes an actor to lose all of its inventory if they are standing on water when this divine attack is done on them Enum class represents different statuses objects in the game can have Abstract class in	**Solution:** An abstract Water class was created that extends Ground and implements the Consumable interface. It is given a WET Status using the addCapability() and Status enum class, and water related grounds such as puddles extend from it. The consume method is made abstract so that it is implemented in its child classes, as this is a very specific method that can have an array of different effects depending on the type of water. I.e. it could poison an actor if it was a poisonous swamp or restore mana if its a puddle. It was logical to make Water an abstract class rather than an interface as it has shared properties and behaviours (like fire resistance and wet status). By using an abstract class, shared statuses and behaviour (e.g. allowable actions()) can be maintained and inherited by subclasses. This implementation was essential for the Frost Divine Power, as it meant that the divineAtttack() method could check if the ground the actor was standing on had the WET status and act accordingly. I.e. an actor will lose all of its inventory if its standing on a wet surface when the DivineBeast uses the Frost Divine Power. **Reasoning:** 10. Each subclass of Water is able to handle its unique behaviours (**Single Responsibility Rule**). , i.e. creature spawning for a puddle, while leveraging shared functionality from the water class. I.e.

	the game that represents wet ground. I.e. a puddle or a Swamp	inheriting the WET status. (**DRY**). 11. Centralising water behaviour means that new Water subclasses can be added without having to modify existing code in the Frost class to make sure its recognised in the divineAttack() method. E.g. if a Bucket ground class is created and extended from the Water abstract class, it will automatically be recognised by the frost class as it will inherit the WET status from the Water class. (**Open/Closed Principle)** Moreover, abstraction from the Action class through the creation of DivineAttackAction, ensures that special features (e.g. damage boosts) can be added for new bosses by extending from DivineAttack without having to modify DivineBeast class (**OCP**) **Limitations:** 5. The abstraction from Ground to create another abstract class Water for subclasses to extend from increases overhead. The complexity level of abstraction may outweigh the improved SRP compliance, and it may cause confusion between developers as the number of classes increases. 6. The current design is quite rigid and inflexible and does not allow room for much development. E.g. if we wanted the actor to remain wet for a number of turns after they stand on a WET surface. A possible future improvement would be for the Water class to hand down a WetStatusEffect to an actor that stands on a wet surface. It was decided against using this implementation, as in the assignment specifics, the Frost class only cares if the actor is standing on the WET ground for the duration of the divine attack therefore it seemed unnecessary. Though, a possible area for development in the future.