Classes Modified/Created	**Roles/Responsibilities**	**Rationale**
Gate (extends item) Application	Class representing a Gate item that actors can use to travel between maps Initiates different GameMaps	**Solution:** The Gate class is implemented by extending the Ground class, inheriting essential attributes like name and symbol. Additionally a new TeleportAction was created that extended Action, with very similar functionality to MoveActorAction, with slight modifications including taking GameMap as a parameter in its construction as teleportation is done across different maps, as well as different toString() functions. **Reasoning:** 1. The Gate class focuses solely on representing a gate object. The logic for transporting an actor from one map to another is handled separately by the MoveActorAction class. This allows the Gate class to encapsulate the representation of only the gate itself, without needing to manage transport mechanics. (**Single Responsibility Principle**) 2. By extending from the Ground class, the Gate can function as a specific type of ground, which allows it to be recognized as its parent class without disrupting overall functionality. For example, in the world class, all classes that extend from Ground, namely Gate, will be recognised in the processActorTurn method when here.getground() is called, without having to make a new method for every type of ground. (**LSP**) 3. Similarly, this improves code reusability and maintainability as high-level modules such as World class don't depend on low-level classes such as Gate and are therefore not affected by their specific changes. Instead it depends on the Ground and Action classes and this abstraction allows for decoupling between high-level and low-level modules. (**Dependency Inversion Principle**) 4. The creation of the addDestination() function allows this implementation to adhere to the **Open/Closed Principle** (**OCP**). Users can easily add new destinations to the gate by simply appending the coordinates of an existing gate using this function, without

		needing to alter any existing code. This design fosters extensibility while ensuring that the original functionality remains intact. 5. Common methods such as toString() are implemented within the Ground class and the Gate subclass is left only to handle gate-specific functionality like transportation, avoiding redundant code. (**DRY**) **Limitations:** 1. Manual Configuration: The current implementation requires users to manually input the destinations. This creates the potential for human error, such as invalid coordinates. 2. While important for a clean design, TeleportAction is quite similar to MoveActor action and a similar and only slightly suboptimal outcome can be achieved using the existing MoveActorAction. The execute method in this class will return that the actor has "moved" rather than "travelled". The excess creation of Action subclasses increases overhead and may lead to confusion between developers when trying to keep track of their intended purpose **Alternative Solution:The** The Gate is responsible for representing the gate and managing the teleportation process. I.e. TeleportAction is not used **Limitations of Alternative Solution:** 1. Violation of SRP → The Gate is responsible for both representing the gate and managing the teleportation process 2. The execute() method always teleports the actor to the first 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**)
PoisonedEffect (Extends Status Effect)	A class representing the poisoned status effect applied to actors who traverse a Poisonswamp.	**Solution** A PoisonSwamp class is implemented by extending the Ground class, applying the PoisonedStatusEffect to any actor standing on it at each game tick. The PoisonedEffect class is implemented by extending the StatuEffect class, managing the number of turners an actor is poisoned and applies 5 damages per tick and is automatically removed from the actor after 3 turns. Extending from Status Effect means that this instantiations of this class will have a

PoisonSwamp	A class representing a poisonous swamp that actors can traverse	concept of time. This separation ensures that PoisonSwamp is only responsible for the application of the Poisoned effect, while the mechanics for handling the poison status is delegated to the PoisonedEffect class. 1. The separation of the two classes ensures that the PoisonSwamp class is only responsible for representing the poisonous terrain and applying the poisonous status effect, while the PoisonedEffect handles the logic of dealing the damage and tracking the number of turns poisoned. This ensures code maintainability and extensibility, so that changes to either class, e.g. modifying the number of turns the damage is applied only has to be handled in the PoisnonedEffect class and the PoisonSwamp class can remain unchanged **(SRP)** 2. Both classes extend from abstract classes, inheriting common ground behaciour while implementing their own specific functionality. This means that any code that interacts with the Ground or StatusEffect class, will work seamlessly with PoisonSwamp and PoisonedEffect simultaneously. For example, in the world class, all classes that extend from Ground, namely PoisonSwamp, will be recognised in the processActorTurn method when here.getground() is called, without having to make a new method for every type of ground. This ensures scalability and extensibility, as new Grounds can be created without having to modify existing code to check for them → they are substitutable with their parent class. **(Liskov Substitution Principle)** 3. Similarly, this improves code reusability and maintainability as high-level modules such as World class don't depend on low-level classes such as PoisonSwamp and are therefore not affected by their specific changes. Instead it depends on the Ground and StatusEffect classes and this abstraction allows for decoupling between high-level and low-level modules. (**Dependency Inversion Principle**) 4. PoisonSwamp only is responsible for implementing methods that are necessary for its specific function, and it inherits common methods such as setDisplayChar(), to avoid redundancy code and unnecessary repetition of existing code (DRY) **Limitations:** 1. Limited extensibility of Poison Logic → the PoisonedEffect class is limited to applying 5

		damage per tick for 3 turns. Adding new variations of poisoned with different attributes would require creating new classes that StatusEffect with different hardcoded attributes, which may lead to extensive overhead and confusion if there are a lot of different Effect classes, as well as duplicate code if developers are not careful **Alternative Solution:The** The PoisonSwamp class is responsible for representing the gate and managing the PoisonEffect process. I.e. PoisonedEffect is not used **Limitations of Alternative Solution:** 3. Violation of SRP → The PoisonSwamp is responsible for both representing the gate and managing the dealing with the poison effect process 4. The tick() method always deals the same amount of damage for the same amount of ticks, making this logic rigid and inflexible. This hardcoding doesn't allow for multiple different effects to be added to the actor (e.g. sickEffect if they are mostly poison resistant but still feel nauseous) when they are standing on the puddle, without having to modify existing code, which could break other parts of the problem (violates **OCP**)