Assignment 1 Design Rationale

REQ 1

The system introduces a game with a Broadsword item that has both active and passive abilities. The player can wield the Broadsword, which grants passive bonuses and an active skill called Focus. DRY is applied by centralizing the logic for the Broadsword's active and passive abilities within the Broadsword and Focus classes. This approach reduces redundancy and makes it easier to maintain and modify the behavior of the Broadsword and its skills. While DRY promotes code clarity and maintainability, it may lead to more complex class structures. In this system, it requires the creation of several classes and their interactions to implement the desired behavior. However, the benefits of reduced redundancy outweigh this drawback.

The system utilizes loose coupling; the Broadsword, Focus, and Player classes interact through well-defined interfaces, reducing dependencies between them. This allows for easier maintenance and extensibility. Each class has a clear and focused responsibility. The Broadsword class handles weapon attributes, while the Focus class manages the activation and deactivation of the skill. This high cohesion ensures that each class has a single, well-defined purpose.

The design choice to use both active and passive abilities enhances gameplay complexity and strategy creates a more engaging gaming experience. The inclusion of both active and passive abilities adds complexity to the system, which can make it harder to understand and test. However, the benefits of enhanced gameplay outweigh these drawbacks.

Active Abilities: The Focus active skill adds depth to gameplay by allowing the player to make strategic decisions about when to activate it. The extension of Action in PerformSkillAction is created in the process of following SRP to provide Player with actions to perform an active skill. Player has a dependency on Action to check if they are available in the CapabilitySet.

Passive Abilities: The passive ability of the Broadsword is triggered automatically when the Broadsword is in the player's inventory. This provides a straightforward way to apply bonuses without requiring active player involvement.

REQ 2

Both Void and Graveyard extend the Ground class. This design choice aligns with the concept of inheritance. Inheritance allows for code reuse and promotes the DRY principle. By extending the Ground class, its properties and methods are inherited, reducing code duplication and ensuring consistency.

The Void class represents a ground type denoted by the character '+'. Its primary purpose is to interact with actors on the game map, specifically rendering them unconscious when they step on it. This choice of behavior aligns with the cohesion principle, as Void has a single, well-defined responsibility related to actor interaction.In the tick method, if an actor is present on the Void, it calls the unconscious method on the actor and checks if the actor is conscious. This design follows the principles of low coupling and high cohesion, as the class only interacts with its contained actor and does not have external dependencies.

The Graveyard class represents a ground type denoted by the character 'n'. Its primary purpose is to have a chance of spawning a WanderingUndead actor when an actor is present at that location. In the spawn method, a random percentage is calculated, and if it falls below a predefined chance threshold, a WanderingUndead actor is added to the game map.

REQ 3

Change is made to WanderBehaviour to make sure it will always return an attack action if there are enemies nearby. The floor is now inaccessible to actors other than player by checking the display character.

REQ 4

The Gate class abstracts the concept of a teleportation gate between two game maps. It encapsulates the origin and destination maps, coordinates, and the ability required to use the gate. This design adheres to the SRP as the Gate class has a single reason to change: when the teleportation behavior needs modification. The Gate and OldKey classes inherit from the base Item class. This design follows the Open/Closed Principle because new types of items or gates can be added without modifying existing code. For instance, you can easily add more types of keys or gates with different abilities without altering the core logic.

The Gate class also encapsulates the teleportation logic, ensuring that it is not scattered throughout the codebase. This adheres to the DRY principle, as teleportation is defined in one place, making it easy to modify and maintain. The Ability enum is used to represent the required key to unlock a gate. This design decision promotes code readability and scalability. For instance, if more abilities are introduced in the game, you can easily extend the enum without altering existing code.

The code is designed to be easily extendable. New game maps, gates, or keys with different abilities can be added without significant changes to existing classes. This follows the Open/Closed Principle as the code is open for extension but closed for modification. It also handles scenarios where an actor attempts to use a gate without the required ability. It checks for the player's capability before teleporting, preventing unauthorized access.

REQ 5

There are two distinct maps in the game world, the Abandoned Village and the Burial Ground. In the maps are different enemies: Wandering Undead and Hollow Soldier. These enemy characters are placed in the game world and serve as adversaries that the player can attack and loot the spoils that they left when they die, such as Key, Healing Vial and Refreshing Flask. These items or spoils can be picked up by the Player and with it they return a Replenishing Action. When the action is chosen by the Player to be executed, player's health or stamina will be regenerated by a certain amount. However, these items are not guaranteed to appear when the enemy dies, only by chance.

In terms of concepts and theories, the DRY principle has been followed by encapsulating similar functionalities into reusable components. For example, the Replenish Action class is designed to handle health and stamina replenishment for various items. Instead of duplicating code for each item, we have a single Replenish Action class that can be reused by multiple items with different

parameters. Replenish Action class also has low coupling because it interacts with the Actor and Item classes without relying on unnecessary external dependencies. Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. The proposed system follows a design where high-level modules for example, actors interact with low-level modules such as items through abstractions like the Action class.