Requirement 1:

The player class should be responsible for handling the player's attributes like the hitpoints and the stamina, the players actions like attacking and moving, the weapons the player can use like the intrinsic weapon and weapons it can pick up.
The Broadsword class is a weapon with a Focus Skill that works for 5 turns. The weapon's skill timer, damage, hit rate, damage multiplier and the weapon reset are all handled by the Broadsword class. The Focus Class handles all the special effects that have to be added to the weapon. It handles the skill duration and all the upgrades to the weapons stats.
This adheres to the Single Responsibility Principle since player class and the Broadsword class will take care of all the tasks and attributes related to themselves.

An alternative way to make the Focus Class would be to assign the skill duration to the Weapon Itself so it doesn't create any unnecessary associations with other classes. But this method violates the Single Responsibility Principle since we are not allowing the Focus Action to assign its own duration.

The Player and the Broadsword has 2 constructors, one with the default values for all its attributes and one for when we need to assign different values to its attributes. The weaponSkill interface can be used for any weapon that implements a type of weapon skill. Focus can be achieved on any weapon that decides to add it to its action list since it is independent of the weapon used. All of these factors make the code Open to extension but Closed to modification

Since Broadsword is just a weapon item it can polymorph into a weapon item or item and perform any of the functions that a weapon item or item should perform. The broadsword can also polymorph into the Weapon Skill interface that we use in the Focus Action to perform a function that is unique to all weapons that implement weapon skill. All these prove the Liskov Substitution Principle.

The interface segregation principle states that classes should not be forced to depend on interfaces that they don't use. Weapon Skill is a weapon specific skill that only needs to be implemented if the weapon has a special ability or skill. Any weapon that doesn't have a weapon skill will not be forced to initiate a contract with the interface.

High level classes like World do not depend on low level classes like Broadsword to function normally, they depend on high level abstractions like item or weapon item to function. This ensures consistent behaviour of our application regardless of the functionality of low level classes. This adheres to the Dependency Inversion Principle.

Requirement 2:

The void class needs to kill any actor that steps on it, we handle the killing of all the actors in the void class itself. The graveyard class runs the spawnactor method of the spawner given to it and controls the spawning of the actor all on its own. This adheres to the Single Responsibility Principle since both these classes handle their own responsibilities.

Any new actor that is added to the game can always polymorph into an actor so the void class can deal with any actor that is added to the game. If we want a new type of graveyard that spawns a new type of actor all we have to do is create a new spawner that spawns a different type of actor and pass that into the graveyard's constructor. These additions make the void and the graveyard classes open to extension but closed to modification since we do not need to modify anything.

The Liskov Substitution Principle is followed due to the fact that any new actor or entity will always polymorph into an abstract form to interact with these classes. This will make it easily and seamlessly integrated into the game.

Since spawners only form contracts with types of spawners, the system doesn't cause classes to unnecessarily create contracts and implement methods. This leads to prevention of unnecessary dependencies between the classes. This adheres to Interface Segregation Principle.

All classes that we have created and implemented have abstract base classes or interfaces. This causes high level components to only interact with high level classes and vice versa. This leads to the Dependency Inversion Principle.

Requirement 3:

The Wandering Undead class handles all the attributes and functions (eg: Intrinsic Weapon) related specifically to the wandering undead itself. This follows the Single Responsibility Principle.

The wandering undead has 2 constructors which leads to less modification of the wandering undead class if we want the wandering undead to have different attributes. This means the wandering undead class is closed to modification but open to extension

Since the wandering undead extends from an enemy class which then extends an actor class that means it can polymorphically act like an actor or an enemy. This leads to the wandering undead being able to pretend to be an actor or an enemy in any method or class. This means that the class utilises the Liskov Substitution Principle.

Wandering Undead extends Enemy which itself extends Actor which means that all high level logic can work with high level classes like actor and enemy which prevents any integration problems. This follows the Dependency Inversion principle.

Requirement 4:

Both the Gate and Old Key classes adhere to the Single Responsibility Principle by maintaining a clear focus on their specific roles. The Gate class has a single responsibility, managing the gate's state and behavior, including checking if it's locked or unlocked and handling player travel. Similarly, the Old Key class maintains a clear focus by representing the key item in the game.

The design also follows the Open-Closed Principle, allowing for the extension of gate types and key items without modifying the existing code. New gate types or key items can be introduced through subclassing or creating new classes, ensuring that the Gate and Old Key classes are closed to modification but open to extension.

Furthermore, the Gate class's ability to handle different types of gates and the Old Key class's capability to represent various key items in the game align with the Liskov Substitution Principle. This design allows for the seamless integration of new gate or key types without causing compatibility issues.

Lastly, high-level game logic interacts with the Gate and Old Key classes through abstract interfaces or base classes, adhering to the Dependency Inversion Principle. This decouples core game logic from specific gate and key implementations, promoting flexibility and maintainability within the game architecture.

Burial Ground Map and Travel Mechanism Rationale:

The Burial Ground Map class strictly adheres to the Single Responsibility Principle by focusing solely on its primary responsibility - representing a new game region and providing the environment for player exploration.

In alignment with the Open-Closed Principle, the Burial Ground Map class is intentionally designed to be open for extension. This design choice allows for the addition of new features and functionalities specific to the burial ground region without necessitating alterations to the existing structure of the class. This approach ensures that the Burial Ground Map class remains closed to modification but open to extension, facilitating the seamless integration of new content.

Additionally, the design supports the Liskov Substitution Principle by enabling the introduction of new game regions as subclasses or implementations of the Burial Ground Map class. This approach ensures that different regions can be smoothly integrated into the game without causing compatibility or integration issues.

Moreover, high-level game logic interacts with the Burial Ground Map class through abstract interfaces or base classes, strictly adhering to the Dependency Inversion Principle. This design practice promotes flexibility in map design, allowing for various map implementations, while simultaneously ensuring that the core game logic remains decoupled from specific map details. This enhances the overall flexibility and maintainability of the game architecture.

Requirement 5:
The Hollow Soldier class embodies the Single Responsibility Principle, as it manages all attributes and actions specific to the Hollow Soldier enemy type. This includes attributes for hit points, attack damage, and accuracy, as well as handling enemy attacks on the player.

In accordance with the Open-Closed Principle, the game's design remains extensible. New enemy types, items, or drop probabilities can be seamlessly introduced without altering existing code. This extensibility allows for the addition of fresh content through subclassing or the creation of new classes, preserving the closed-to-modification but open-to-extension paradigm.

Moreover, the Hollow Soldier class aligns with the Liskov Substitution Principle, enabling it to function as an enemy type within the game. This versatility allows for the effortless integration of diverse enemy types without causing compatibility issues.

High-level game logic interacts with the Hollow Soldier class through abstract interfaces or base classes, upholding the Dependency Inversion Principle. This design approach decouples the core game logic from specific enemy implementations, thereby enhancing overall flexibility and maintainability.

The design choice to calculate the probability of item drops independently of each other enhances gameplay variety. This means that, after defeating a Hollow Soldier, players have the potential to receive both healing vials and refreshing flasks, introducing diversity in item drops and enriching the player experience.

Furthermore, players must pick up healing vials and refreshing flasks before consumption. This reinforces the game's mechanics and introduces an additional layer of player interaction.

Consistency in item drop mechanics is maintained by allowing the Wandering Undead enemy in the Abandoned Village map to also have a chance to drop healing vials. Importantly, the probability calculation for the healing vial drop remains independent of the Old Key drop.