Design Rationale

Requirement 1

1. In the Player class, capabilities are implemented using a combination of string constants (e.g., "Status.HOSTILE_TO_ENEMY") and the addCapability method. Capabilities are added to a Player instance during its construction, allowing us to specify which capabilities the player possesses. These capabilities can then be checked using methods like hasCapability, enabling the Player class to respond differently based on its characteristics.
    a. Pros
        i. Actors can have flexible and extensible features thanks to capabilities without changing the actor class. This follows the Open-Closed Principle, which dictates that classes should be open for extension but closed for modification. This is significant because it allows the core implementation of actors to change without introducing new capabilities. It also eliminates the need for multiple instanceof checks, as we can simply look for actors with similar capabilities.
        ii. Multiple actor classes (such as various opponents or characters) may have similar skills or ability in the game. These classes can implement or inherit shared behaviours using capabilities without adding extra classes or duplicating code. This allows the prevention of using multiple if…else statements to check for an instance of a class as we can establish each of these classes with common capabilities
    b. Cons
        i. As the number of players and capabilities rises, managing several capabilities for an actor can become challenging.
            - However, this trade-off is preferable to using instanceof type checks, which can lead to tight coupling between classes and make the code harder to maintain.

2. This Java code implements a game system with a weapon, the "Broadsword," that can activate a Focus Skill for a limited number of turns to increase its hit rate and damage. It uses an interface called WeaponFocus to define methods for active and inactive focus states. This approach allows the implementation of different weapons' state changes when the Focus Skill is applied to be controlled by the weapon itself. Different weapons with the same focus skill might have different increases or decreases in hit points, damage multipliers, or even damage. The Broadsword class extends a WeaponItem and implements WeaponFocus and overrides tick methods to check for the number of turns for the FocusAction and reset the state when Broadsword on ground, while the FocusAction class controls the activation and deactivation of the focus skill by storing the weapon as a WeaponFocus class. This affects the weapon's properties (controlled by the weapon itself) and consumes the player's stamina (controlled by the FocusAction class).
    a. Pros
        i. The FocusAction behavior of the Broadsword is contained within a separate class, making the code more comprehensible and modular. The FocusAction class neatly encapsulates the focus skill's logic, which improves code readability and extendibility. Since the FocusAction does not have a specific

implementation of the Broadsword, and the changes of the Broadsword are accessed by the methods in Broadsword (i.e., activeWeaponChange), this design adheres to the Single Responsibility Principle. Different or new weapons can still use the FocusAction implementation by implementing WeaponFocus because the Broadsword's implementation will not affect the implementation of the new weapon. This assumes that the player's stamina reduces by 20% whenever using the Focus Skill, regardless of the weapon.

ii. Since FocusAction is an action that only runs once, I decided to use the tick method to control the number of turns that has passed and allows the Broadsword to be rest if the Max Turns reached (true returned from FocusAction). This maintains the modularity again as each attributes of the classes depends on the respective classes

b. Cons

i. The design may seem a little complex as it involves interfaces, abstract classes, and concrete classes.

- However, since the implementation of my code establishes a better correlation between both FocusAction and Broadsword by having specific implementations for each other, the complexity is a reasonable trade-off for the modularity and extendibility of the system

3. For the playturn of Player, the code starts by displaying information about the actor's name, health, and stamina. It uses the display object to print this information to the game console. It checks if the last action taken by the actor has a next action (lastAction.getNextAction()). If there is a next action, it returns that action. Then, the code calculates and increases the actor's stamina based on a percentage of its maximum stamina. Finally, the code creates a Menu object and shows a menu to the player, likely allowing them to choose from a list of available actions. The menu is generated from the provided actions list. The pros of this code is that it is very manageable and explanatory since the User choosing their options is handled by the Menu class separately allowing Single Responsibility Principle to be adhered. As for the cons, I don’t think there is any as of the implementation right now

Requirement 2

1. As for the adding of the FancyMessage “YOU DIED” when a player dies, I override the unconscious method of the Player to remove the actor, then print out the “YOU DIED” line.
    a. Pros
        i. The primary reason for this approach is that the "YOU DIED" printing statement is specific to the Player, and only the player should display this message when they die. This aligns with the Single Responsibility Principle, as the responsibility for actions when the player dies rests with the player class. Therefore, overriding and printing the "YOU DIED" message in the Player class is a suitable choice.
    b. Cons
        i. If it is necessary to display a similar message for another object (i.e. WanderingUndead), the existing method would not be reusable because it directly binds the display of the death message to the Player class.
            - However, since the Player is the only playable character and the only entity that should print this message, this method is deemed sufficient for the message display.

2. For the implementation of Void, I added a capability for the Void, giving it a status of Status.UNSPAWNABLE. This ensures that any Graveyard that interacts with an object having this capability will not be able to spawn an enemy on that object. The pros and cons are similar to the explanation given in Requirement 1.1. In simple terms, the pros include the ability to implement shared behaviours for multiple ground classes with similar skills or abilities using capabilities, reducing the need for additional classes or code duplication. This helps prevent the use of multiple if...else statements to check for class instances and maintains the Open-Closed Principle. As for the cons, managing multiple capabilities for an actor can become complex as their number increases, but it is still a reasonable trade-off to maintain the Open-Closed Principle.

3. In the implementation of the Graveyard, I chose to make the Graveyard class a child class of the ground abstract class. In my implementation, the Graveyard has a new constructor that accepts an object of the SpawnActor instance. SpawnActor is an interface implemented by WanderingUndead, providing a method for creating new instances of the respective actor. Therefore, whenever a new enemy is spawned, the instance of the enemy specific to the map (passed through the constructor of the Application class) can be used to create new instances of the respective enemy.
    a. Pros
        i. The implementation of Graveyard, SpawnActor, and WanderingUndead allows for modularity and separation of responsibilities. The Graveyard manages actor spawning at a location, SpawnActor defines the spawning behavior, and WanderingUndead manages the type of enemy to be spawned. Thus, having SpawnActor interface that generates a new instance of an enemy based on the specified percentage allows for modularization. This enhances modularity and maintainability, as each class has a single responsibility, making the code easier to understand and maintain.
        ii. This design can be used to address issues related to the Open-Closed Principle, as the spawning of each actor is specific to each enemy type and is

controlled by the respective enemy class. Therefore, for future extensions and the addition of more enemies, there is no need for multiple 'if...else' checks to determine the type of enemy being spawned and to create instances. Instead, the creation of the enemy is controlled by the respective child classes of Actor.

b. Cons
    i. If multiple classes in the game require actor spawning, there might be a risk of code duplication. Actor spawning might be similar across different classes, leading to potential redundancy.
        - However, in this case, actor spawning is specific to the respective enemy type being initialized and spawning also depends on the probability of occurrence set in the Graveyard. Therefore, the likelihood of code duplication is minimal, as each enemy class instantiates itself based on the specific probabilities and parameters, reducing redundancy.

Requirement 3

1. For this requirement, I created the Attack and Wander interfaces to define behaviours related to attacking and wandering. These interfaces serve as a way to encapsulate and abstract these behaviours, allowing me to implement different behavior strategies for the WanderingUndead class. The implementation provides a common contract for any class that implements it to specify how an actor (in this case, the WanderingUndead) should behave when it's in an attack mode or wander mode. The behaviours are also added to the HashMap in the order they are called in the constructor, as both 'wanderBehaviour()' and 'attackBehaviour()' are used to create an instance of their respective Behaviours.
    a. Pros
        i. Instead of creating a single common interface for Behaviours, which would go against the Interface Segregation Principle (an interface that has all abstract methods, even when certain classes do not require them), Attack and Wander being distinct interfaces enable a more adaptable and flexible method of setting behaviours. It allows actors like WanderingUndead to implement the specific behaviours they want in the order they prefer, as these behaviours are specific to the actor classes.
        ii. Looking from the perspective of future extensions of this project, it is unlikely for two different enemies to have the same behavioral changes. By having WanderingUndead implement these interfaces, each can control its own behaviours, making it easier to accommodate unique behaviours for different enemy types. While there may be some code repetition for now, this approach ensures adaptability for future changes.
        iii. By avoiding the establishment of a single abstract class that might not be completely utilised by all opponent kinds, this method lowers the danger of creating excessive complexity. It keeps the codebase simple.
    b. Cons
        i. This strategy allows for flexibility, but because equivalent behaviours must be implemented independently for each enemy class, it may result in some code duplication in the short run.
            - However, the benefits of modularity and adaptability typically outweigh this trade-off in code duplication, as the modularity and encapsulation are maintained.

2. In the implementation of the AttackingBehaviour class, it captures an actor's tendency to attack another player nearby one of its 8 exits. If a player with the Status.HOSTILE_TO_ENEMY capability is found in the area immediately around the actor's current position on the game map, this behavior produces and returns an AttackAction to attack that player. It returns null if no adjacent players are discovered. The pros of this method are that, since this method only depicts the attacking structure of an Enemy if there is a player with one of its exits, this method is flexible and extendable. Other enemy instances, other than WanderingUndead, can also use this behavior, as it is only based on the behavior, not the type of Enemy. As for the cons, I don’t think there are any issues with the implementation as it stands.

3. For the implementation of the Floor class, I created a class that extends the Ground abstract class. Since it can only allow the Player to enter, the 'canActorEnter()' method was

overridden such that actors with the capability 'Ability.ENTER_FLOOR' can enter the floor ground. Thus, actors that want to enter the floor (in our case, only the Player) must add the capability 'Ability.ENTER_FLOOR' to themselves. The pros are that we don’t need multiple if…else checks here, as this would go against the Open-Closed Principle. Additionally, it is extendible, as multiple different players that might have the capability to enter the floor can simply add 'Ability.ENTER_FLOOR' to themselves, while the code in the Floor class remains unchanged, avoiding the need for additional instance checks. As for the cons, I don’t think there are any issues with the implementation as it stands.

Requirement 4

1. For the implementation of the LockedGate, I have designed the LockedGate class as a child class of the Ground abstract class. In this implementation, the LockedGate class has a new constructor that accepts a MoveActorAction as a parameter. If the actor surrounding the Graveyard is a Player (has Ability.UNLOCK_DOOR), they will be able to unlock the gate using the UnlockDoorAction (which adds the Ability.MOVE_MAP to the LockedGate if executed) and move to the next map once it's unlocked.
    a. Pros
        i. This implementation follows the principle of modularity, where the action of unlocking the door is independent of other classes, and both the unlocking and moving actions are encapsulated within the LockedGate class. This separation of responsibilities makes the code easier to maintain and extend
        ii. Adding the Ability to Move Map (as an ENUM) when the door is unlocked also allows us to maintain the state of LockedGate class to check if it is locked or not and also if actors can step on it or not easily. This enhances code clarity and simplifies state management.
    b. Cons
        i. Creating distinct classes for gates and unlocking them could increase complexity, which might confuse people working on the code.
            - Although the complexity might increase, the code maintainability is much simpler as the LockedGate does not have to do all implementation by itself instead it organizes its actions to conduct through an Action instance
        ii. The usage of a parameterized LockedGate constructor that accepts MoveActorAction instance allows flexibility, but it may result in creation and management of numerous action instances, resulting in more complex code.
            - Even if the code seems a little complicated, this implementation is better than creating separate LockedGate classes for each of the Map as this does not follow the DRY principle and increase the complexity of the code as it makes managing these gates much harder because we have to keep track of a larger number of classes.

2. For the implementation of LockedGate since it can only allow actors to step on the ground after it is unlocked (When the player can Move Map as the gate will be unlocked at the same time), the ‘canActorEnter()’ method was overridden such that actors can only enter the LockedGate if it has ‘Ability.MOVE_MAP’. The pros is that it is easier to maintain the code as it follows the principle of the question with minimal code redundancy. As for the cons, I don’t think there is any as of the implementation right now.

3. Based on the implementation of Locked Gate, only the actors with the 'Ability.INTERACT_WITH_GATE' (Player) are able to interact with the gate. Thus, I have added the capability 'Ability.INTERACT_WITH_GATE' to the Player class so that the player is able to interact with the gate so that it can either unlock the door or move the map. Again, the pros and cons are similar to the explanation given in Requirement 1.1. In simple words, the pros are that multiple actor classes with similar skills or abilities can implement or inherit shared behaviours using capabilities, reducing the need for additional classes or code duplication, which helps prevent the use of multiple if...else statements to check for class

instances and maintain the Open-closed Principle. As for the cons, managing multiple capabilities for an actor can become complex as their number increases, but it is still a good trade-off to maintain OCP.

4. For the implementation of OldKey, I have created a new parent class which is an abstract class called DroppableItem that extends Items which has methods to drop an item at the location of an actor.
    a. Pros
        i. Having an abstract class to control the dropping of an item on to a certain location abides by the DRY principle. This allows the sub-classes to not have to repeat the common implementation of the dropping of the item.
        ii. Future extension – if we want each Item dropped when an enemy is killed to have a certain method respective of its class for its implementation, we can just create the method signature (without implementation) in the DroppableItem abstract class which ensures that all the new methods in DroppableItem subclasses is implemented. This ensures that classes have consistent behaviour, making it easier to swap out one implementation for another without affecting the rest of the code.
        iii. By setting the DroppableItem as the abstract parent class, I can prevent concrete classes to have dependency with another concrete class (i.e. WanderingUndead with OldKey). This allows WanderingUndead to depend on an abstract class which is DroppableItem which is implemented by OldKey. This follows the last solid principle, which is Dependency Inversion Principle
    b. Cons
        i. Having an abstract class might cause limited flexibility as it imposes a structure where all the behaviours of the parent class must be extended by the child classes (even if the implementation is not required)
            - While abstract classes ensure shared implementation, interfaces provide more flexibility when common implementation is not required. So, in our case we can use interfaces for items with unique behaviours while reserving abstract classes for items with common actions such as dropping an item at a location which is common for each object

5. For the current implementation of DroppableItem, it only handles the common process of dropping items onto a location. However, we can still utilize interfaces to implement specific methods for different DroppableItems, such as OldKey or others that may be added in the future. To achieve this, I've enhanced these classes by introducing interfaces in the WanderingUndead class. Specifically, I've created an interface named 'DropOldKey,' which the WanderingUndead class implements. This approach ensures that WanderingUndead will provide an implementation for the 'dropOldKey' method. An interface is chosen because the way keys are dropped may vary among enemies; some may drop multiple keys, while others may drop just one. This approach also supports extensibility.
    a. Pros
        i. The pros of this approach include the promotion of modularity in the code. If there's a need to introduce more types of items that can be dropped by WanderingUndead or other actors in the future, we can simply create new

interfaces and have the relevant classes implement them, all without impacting existing code.

b. Cons
    i. The cons involve the introduction of multiple interfaces, potentially increasing code complexity. However, this increased complexity is a reasonable trade-off given the benefits of this implementation. It's a superior approach compared to forcing all WanderingUndead to implement all possible DroppableItems, even when such implementations are unnecessary.

Requirement 5

1. For the implementation of HealingVial and RefreshingFlask, since we have already implemented an extensible DroppableItem framework, we will utilize the same approach here. However, since each class can have multiple DroppableItems (for instance, WanderingUndead can have OldKey and HealingVial), we will modify our implementation to maintain a list of DroppableItems. When an actor becomes unconscious, it will iterate through each potential DroppableItem to check if it can drop the item based on the randomly calculated probability value.
    a. Pros
        i. This technique simplifies future expansion of droppable items without altering the existing code. We can easily create new classes that extend the DroppableItem abstract class and add new types of items to the list of items that can be dropped when an actor is unconscious.
        ii. The code's modularity improves because each DroppableItem is encapsulated within a separate class. This promotes the separation of responsibilities and makes the code more readable and maintainable.
    b. Cons
        i. The complexity of the code can rise as a result of managing a list of droppable things and determining their drop probabilities, particularly as the number of actors and droppable items rises. The intricacy may make it more difficult to maintain and troubleshoot the code.

2. For the HollowSoldier class, even though it exhibits similar behaviors such as attacking or wandering (having extensibility for both Attack and Wander interfaces and containing instances of both Attacking and Wandering Behaviors), I have opted not to create a new abstract Enemy class that would extend from Actor, a structure both WanderingUndead and HollowSoldier would inherit. Instead, I have maintained the implementation of the Attack and Wander interfaces.
    a. Pros
        i. Avoiding the creation of a single common interface for behaviours, which could violate the Interface Segregation Principle (an interface with all abstract methods even when certain classes do not require them), enables a more adaptable and flexible approach to setting behaviours. By avoiding unnecessary code inheritance and potential conflicts, it allows actors like WanderingUndead and HollowSoldier to implement the specific behaviours they require, in the order they prefer, as they are specific to the actor classes.
        ii. From a future extension perspective, it is highly unlikely for two different enemies to share identical behavioural changes. By having WanderingUndead and HollowSoldier implement these interfaces, each can control its own behaviours, making it easier to accommodate unique behaviours for different enemy types. While there may be some code repetition initially, this approach ensures adaptability and extensibility for future changes.
        iii. By avoiding the establishment of a single abstract class that might not be completely utilised by all opponent kinds, this method lowers the danger of creating excessive complexity. It keeps the codebase simple.

b. Cons
    i. This approach allows for flexibility, but because equivalent behaviours must be implemented separately for each enemy class, it may result in some code duplication in the short term. However, the advantages of modularity and adaptability typically outweigh the trade-off in code duplication.