Assignment 1 Design Rationale

Introduction

The implementation of various functionalities within the game is an important design decision, which must follow the SOLID and Object-Oriented Programming principles for robust and maintainable code. This design rationale discusses the implementation of Designbourne game features while highlighting how they align with these principles.

Requirement 1

Player's stamina

Initially Player had only one attribute which is their health, so the first feature implemented in the game was the player's stamina by using BaseActorAttribute class and adding a new element in the BaseActorAttributes enumeration. The main advantage of this implementation is that it follows the Open/Closed principle since it has extended the player's functionality without changing existing code much.

Moreover, stamina must be increased by certain percentage in every turn of the game. In order to maintain that a new method recoverStamina() was created inside the Player class and called by playTurn() method. This implementation aligns with the Single Responsibility principle by preventing playTurn() method from becoming overly complicated with additional functionalities. This separation of concerns enhances code maintainability and readability.

Broadsword

The Broadsword weapon was extended from the WeaponItem class. Its constructor accepts parameters such as damage, hit rate, the duration of the Focus skill, as well as an increase in damage multiplier and increase in hit rate while the skill is active. Since the Broadsword has a special skill, it implements an interface AbleToActivateSkill and the method activatedSkill which manages weapon's skill. This aligns with the Interface Segregation principle since weapons that have a special ability can implement this interface, and others do not need it.

The implementation of the Focus ability as an Action helps making it executable for the Player. When action is executed, it deducts 20% of the player's stamina. Action cannot be executed is stamina is lower than 20% of the maximum stamina. Furthermore, it also adds a new ability enumeration to the weapon, indicating that the focus ability is active. After a certain number of turns, this ability is removed from the weapon, restoring it to its initial stats. Also, the Focus skill is removed if the weapon is lying on the ground.

This design choice is perfectly aligning with the Open/Closed principle as the Focus ability is implemented as a separate action and enumeration. This ability can also be easily implemented into other weapons within the game which makes the code extendable. Adding AbleToActivateSkill interface also helps differentiate between weapons with the skill and without. The potential downside is that this implementation adds more abstraction to the code.

Requirement 2

Bottomless pit

Bottomless pit or void is implemented as a ground. If there is any actor on the location of the void, health of the actor is reduced to zero. If in the player's or enemy's turn, their health is zero, DeathAction is called and executed.

DeathAction allows to kill a character if they are unconscious due to any natural causes or accident. It can be used in the future if new features that kills the character are implemented. It aligns with Single Responsibility principle, because this action takes off responsibility for eliminating an actor from the killing events. Potential drawback of this implementation is that both the player and enemy actors check their consciousness at the beginning of each turn. This dual check may result in redundancy and impact performance efficiency.

Graveyard

Abstract graveyard class acts a spawner for the enemy. This class spawns a certain monster based on the given probability. An assumption underlying this design is that a monster can only be spawned if the graveyard is unoccupied by another character. WanderingUndeadGraveyard is extended from the Graveyard abstract class allowing the creation of a unique graveyard instance, which has an ability to spawn the Wandering Undead.

Since abstract Graveyard class takes probability and a monster as the parameters, new graveyards with different monster can be added to the game which adheres Open/Closed principles allowing further extensions and ensures scalability of the game's enemy spawning functionality.

Also, another design aspect of this implementation is the Dependency Inversion principle. The application creates instances of the abstract Graveyard class, rather than its subclasses. It allows for the interchangeability of different graveyard types.

Requirement 3

Wandering Undead

To implement this feature, a new Enemy abstract class is created by inheriting from the Actor abstract class, introducing two key behaviours: wandering and attacking. This abstract class looks for the action to be executed by this enemy in their turn, and adds allowable actions which includes AttackAction with both intrinsic weapon and with another player's weapons. The player must have a status HostileToEnemies to be able to attack. An enemy abstract class has a capability that it cannot access floor implemented inside floor. Wandering Undead is implemented by extending enemy class and implementing Actor's getIntrinsicWeapon method.

A significant advantage of this approach is adherence to the Don't Repeat Yourself principle. Implementing abstract enemy class first helps making enemy class responsible for the enemies with the similar behaviour without redundant code for each of the subclasses. This makes code efficient and consistent across enemy types.

Also, the abstract class's reliance on the Action abstract class rather than its subclasses aligns with the Dependency Inversion principle. This abstraction provides a modularity by reducing dependencies on specific action implementations.

However, there is a potential drawback. If there is a need to introduce an enemy with entirely different behaviour from the existing abstract class, it may require changing the Enemy class or

creating a new enemy class. This does not align with the Open/Closed principle and can limit the code's extendibility.

Requirement 4

Locked Gate

A Gate is extended from Ground abstract class. Initially, these gates are locked, restricting any actor from entering it. The capabilities are used to determine whether a gate is locked or unlocked to minimize complexity of the system. Once the gate is unlocked, only player can use it to teleport while other NPCs can just enter it without moving to another map.

In the first implementation of the Gate, it receives GameMap as a parameter and uses a moving action to teleport a player, which adds more dependencies between classes. Later, it is changed to the second implementation where a moving action is added to the Gate in the application class, reducing dependencies between the GameMap and the Gate objects. This separation of concerns contributes to a cleaner code and aligns with Reduce Dependency principle. Also, as mentioned in the previous requirements the Gate uses Abstract Action class instead of its subclasses, following Dependency Inversion principle.

A disadvantage of this implementation is that all the Gates that are added to the map are locked, so it does not adhere Open/Closed principle. As an example, if the new gate is added to the game and it is initially unlocked, Gate class must be changed.

In order to unlock the gate a new action class is added to the system: UnlockGateAction is extended from the Action class. This action allows user to unlock a gate if they have a key. The advantage of this action class is that it follows a Single Responsibility principle since the Gate is not responsible for unlocking.

Old Key

The player is able to open any gates as long as they have an Old Key in their inventory. The key is reusable and is not deleted from inventory after the use. It is also assumed that there can be multiple Old Keys on the map.

The first potential implementation of this functionality is that the key appears on the ground after the enemy was defeated based on the probability. The logic is that if the defeated enemy has a capability LEAVES_KEY, AttackAction creates an instance of the GateKey and adds it to the map. But this implementation has multiple drawbacks. The most important one is that it breach the Open/Closed principle. For example, it there is a need to add more enemies which spawns other items, AttackAction must check each capability, and the list in the future can be too long. Moreover, it adds unnecessary dependencies between AttackAction and Item subclasses.

The actual implementation was that the key randomly appears inside the enemy's inventory, the assumption is made that an enemy will definitely drop everything from their inventory. This follows Open/Closed principle that is it easily extendable if new enemies are added. A potential disadvantage is that they keys are spawned before the defeating the enemy so this implementation might not follow requirements correctly.

Furthermore, initially random probability was calculated inside each of the Enemy classes, but this disobeys Single Responsibility principle since Enemy class must perform too many actions. A new static Utility class is added which calculates probability to align with the principle.

Requirement 5

Hollow Soldier

Hollow Soldier class is extended from the Enemy class and easily implemented in the system since both Hollow Soldier and Wandering Undead share the same behaviour. This demonstrates adherence to the Open/Closed principle by adding a new enemy type without changing existing code.

Healing Vile and Refreshing Flask

Initially these two items are created separately, this implementation has a huge drawback which is too many dependencies as well as repetitive code. After that an abstract class ConsumableItem is created, it extends from an Item class and adds an ConsumeAction to the list of possible actions on this item. Both Healing Vile and Refreshing Flask are extended from the ConsumableItem class.

When ConsumableItem abstract class is created, it is assumed that any ConsumableItem can have a positive or negative effect on the actor. Both attribute and attribute operation are passed as an argument to the action. This perfectly align with Open/Closed principle since more consumable items can be easily added to the game. But there might be a problem if the item does not have any affect on the player. These items can be dropped by the certain enemy, and this has the same logic as a GateKey in requirement 4.