Name: Desmond Chong Qi Xiang
Student ID: 33338248

Design Rationale Assignment 1

Req 1

Create a WeaponSkill interface for that weapon that has a special skill

Pros:

- Clear Contract:
  The interface can help us to specify what methods a weapon class implementing it should have.
- Polymorphism:
  The interface can help us to achieve polymorphism, especially in ActiveSkillAction Class, We can let the interface help us to handle different types of special skills with respective weapons instead of checking what type of weapon class each time and performing different actions.
- Dependency Inversion Principle(DIP):
  Interface can help us to achieve this principle because any class using “WeaponSkill” does not need to depend on details, they depend on abstractions just like the ActivateSkillAction.

Cons:

- Increase Complexity:
  Any developer that extends this system, need to implement this interface whenever they create a weapon class which have weapon skill. This seemed to increase the job of the developer. But in the long run, this interface can make the system easier to maintain and extend.

Create an ActivateSkillAction extend the Action class and use the WeaponSkill class instance as an attribute

Pros:

- Single Responsibility Principle(SRP):
  We can just do ActivateSkillAction in the Weapon class but we choose to create a separate class to achieve Single Responsibility. The reason is to make the system can be easily extended, we have to make every class have only one responsibility, so a class can focus on doing one thing.
- Reduce the occurrence of logic errors:
  In ActivateSkillAction Class, we use the WeaponSkill instance instead of using the WeaponItem instance because Java will generate a compile error if we try to pass in a WeaponItem instance. This can remind developers if they are passing in a weapon without special skill or forgot to implement the WeaponSkill interface in the weapon class.

Cons:

- Increase Complexity:
  This implementation can make developers use a lot of time to understand how to develop this part of code but in the long run, we can make the system easier to maintain since this way of design can reduce the situation of use a lot of times finding logic error.

Create a TheAbandonedVillage extends GameMap class
Pros:

- Single Responsibility Principle(SRP):
  We separate the TheAbandoneVillage class from GameMap because, in the end, we can extend more features which different from the original GameMap class and use the same feature as GameMapClass by extending it.

Cons:

- Redundant:
  We can create a TheAbandonedVillage object by just using the GameMap class. Creating a separate class seemed redundant for now. But in the future, it is easier to extend the feature of it.

Req 2

Create a Spawner interface and each Enemy Spawner class
Pros:

- Dependency Inversion Principle(DIP):
  The reason for creating an interface is because we want to achieve polymorphism, especially in the Graveyard class which uses the Spawner instance as an attribute. This can make us not create a Graveyard class for each game map. Therefore, the graveyard class does not need to care about which enemy spawner class it is addressing now. This will let the graveyard class depend on abstractions instead of details.
- Open-Closed Principle:
  Because of the interface, in the Graveyard class, we do not need to modify the code when adding a new enemy to the system. We can create a new enemy spawner for that enemy and make that spawner class implement the spawner interface. Thus, the Graveyard class can work with any enemy spawner class that passes in when creating a graveyard class instance.

Cons:

- Increase the number of enemy spawner classes:
  This way of design will increase the number of enemy spawner classes whenever a new enemy type is added. However, it is easier to extend when the system becomes larger instead of using an enemy spawner class to handle all types of enemies.

Create a ProbabilityGenerator class to handle all events that depend on a chance
Pros:

- DRY principles:
  Due to many features depending on chance to occur in this game system, and the difference of the code is just the percentage of probability, we can create a class to handle all chance events and get the result by inputting the percentage of probability.

Cons:

- Redundancy:
  Directly writing code may seem the same as using the class static method because it does not reduce the code too much. But every time, a developer encounters a chance event can just use the method instead of thinking about how to write the code.

Req 3

Create an Enemy abstract class and WanderingUndead class extends it
Pro:

- DRY principle:
  Because many enemies in this game system have the same feature. Therefore, we create an abstract class to avoid the repeating code. Any enemy that is added can extend this class and extend the different features with other enemies.

Cons:

- Open-Closed Principle:
  This way of design may violate the Open-Closed principle in the future when an enemy has a different feature from the original enemy class. However, we can modify the different parts in the Enemy class. For now, we chose to sacrifice this to achieve the DRY principle.

Adding the ability to enter the floor to the player and making the floor class only allows the actor who has entered the floor ability.
Pro:

- Open-Closed Principle:
  Instead of checking whether the type of actor entering into floor is a player or not, we check whether the actor can enter the floor. Therefore, the code on the floor will not be modified. Whenever an actor has the ability, we just need to add the enter floor enumeration to the ability of the actor.

Cons:

- Increase complexity:
  Whenever an actor can enter the floor, we must update the actor attributes. This may seem like a code smell. But it is a better way instead of updating the floor class every actor is added.

Req 4

Create UnlockGateAction class extends Action and unlocks the gate only if the actor has the key by checking the actor has the enumeration ability.
Pros:

- Open-Closed Principle:
  The intention of doing this is the same as the req3 implementation. Instead of checking all items in the actor inventory and finding out the key instance using an instance of method which is a code smell. We updated the key class to add the capacity of the key. Therefore, whenever any other new item class can unlock the gate other than the key, we do not need to modify the code in UnlockGateAction.

Cons:

- Increase complexity:
  Whenever an item has an unlocked gate ability, we need to update the ability in the constructor. This implementation may make another developer confused and use time to find out. But it makes the system easier to maintain in the long run.

Req 5

Create a Consumable interface, make HealingVial and RefreshingFlask class implement it, and use the Consumable interface as an attribute in ConsumeAction

Pros:

- Dependency Inversion Principle(DIP):
  Consumable interfaces help us to depend on abstractions rather than concrete implementations. The ConsumeAction class uses interface as an attribute to achieve polymorphism without care about which item it is dealing with. Therefore, we do not need to create ConsumeHealingVial and ConsumeRefreshingFlask classes to have different implementations. This helps in reducing tight coupling between different parts of a system, making it more flexible and easier to maintain.

Cons:

- Single Responsibility Principle (SRP):
  This implementation seems to violate the SRP principle since we do not separate the ConsumeAction corresponding to a different item. But the implementation is the same as creating an interface because we let the interface help us to separate responsibility.