The rationale for Zixin's part UML class diagram

Requirement 1-(Player and Estus Flask):

As required, I decided only to create **HealAction** class to implement the two steps of the requirement1.

Design reason:
Because healing the player himself is an action, other sub-requirements can be implemented within the **Player** class as attributes or methods(like **HealthPotion** can be considered as an attribute, show player's information can be done by overriding toString()). It meets the single responsibility principle. **HealAction** has an association relationship with the **PlayerInterface**. That's because the **PlayerInterface** is a variable("target") in the **HealAction** class. The benefit of the design is that it conforms with the "interface segregation principle" and "Dependency inversion principle".

Requirement 2 (Bonfire):

Bonfire class and **Vendor** class inherited the **Ground** abstract class. Because they don't need almost all functions that actors have, and inheriting **Ground** is suitable for them. Bonfire has an action called "rest", so it has a dependency relationship with the **RestAction**. **RestAction** could refill the player's hit points and Estus Flask, which has an association relationship with the player's interface.

Requirement 3 (Souls):

The Player has an implementation relationship with the **Souls** interface so that that player can add or subtract his souls-number. **TradeWeaponAction** and **AttackAction** could directly use the player's interface and methods on the **Souls** interface to implement those requirements. It follows the "Dependency inversion principle" to reduce coupling.

Requirement 4(part of enemies):

(only for my part)

Skeleton and undead both are enemies, so there are some repeating functions or attributes. To make the code try, they inherited the **Enemy** abstract class. Each enemy has different abilities and status, so they have a dependency relationship with those two enums (**Ability** and **Status**). Skeleton holds a random weapon at the beginning, so the **Skeleton** class should have an association relationship with the **Weapon** interface (It follows the "Open/closed principle" since Weapon interface has some children class including those two weapons that enemy can hold. It also follows "Dependency inversion" because any specific weapon class changes will not directly influence the **Skeleton** class --using the interface).

I create a **ResurrectBehaviour** class, and it will contain resurgence action because of

the skeleton's skill. **ResurrectBehaviour** implements the Behavior interface. **ResurgenceAction** inherits the Action class and has an association relationship with the skeleton's interface because the action needs to know the target(skeleton itself). Finally, the skeleton could use the behaviour to do its skill, so there is a relationship between the Skeleton class and **ResurrectBehaviour** class. The design reason is using behaviours allows us to modularize the code that decides what to do, and that means that it can be reused if more than one kind of Actor needs to be able to resurrect themselves in future.

For **FollowBehaviour** and **WanderBehaviour**, every enemy has those two behaviours, so I coded the same relative code on the parent abstract class (Enemy) to make the code dry. That's why the Enemy class has a dependency relationship with those two behaviours.

Requirement 5 (Terrains):

Valley and cemetery both have a kind of ability, so they have a relationship with the Abilities class. And valley will kill the player if he/she is on the top of the terrain (valley has to know who is the target(player)), so it has an association relationship with the **PlayerInterface**.(Valley could use interface to revoke methods that belong to the player)

Cemeteries will generate undead with the help of **CemeteryHelper**'s Interface. First, I create **CemeteryHelper** to replace all cemeteries with the new cemetery that contains **CemeteryHelper**'s interface. After that, the new cemeteries will know their **Location** and create undead on the game map. Because cemeteries have to know their **Location** on the game map to create undead instances automatically at each turn, I cannot pass the game map to each cemetery without changing the code in the engine folder since cemeteries are created before creating the game map.

Requirement 6:

(there is no my work about it)

Requirement 7(Weapons):

(only for my work - Broadsword class, Giant Axe class, Player Intrinsic Weapon class)

Broadsword and **GiantAxe** implements the abstract **MeleeWeapon** class, and **MeleeWeapon** class inherits the **WeaponItem** class. The design reason is that they both have the same attributes(like price) and similar limitations (skeletons

can only hold these two kinds of weapons, so **Skeleton** needs to make an association relationship with the **MeleeWeapon** class). The **PlyerIntrinsicWeapon** class inherits **IntrinsicWeapon**, which customizes the player's intrinsic weapon.

Requirement 8:

Vendor class will use **TradeWeaponAction** (dependency between vendor and **TradeWeaponAction**) to make a deal with the player (association between vendor and player's interface), and the vendor also needs to use the **Broadsword** class and the **GiantAxe** class (dependency between them).