UML class diagram rationale:

(Some exist features are not shown and classes name is bold on the following rationale)

I am responsible for the class diagram part. I incorporated design principles such as “reduce dependencies”, “don't repeat yourself”, “command-query separation in my preliminary design”, “classes should be responsible for their own properties” and will consider other principles in my implementation.

Overall, I divided these classes into different packages. It makes it easier to find related classes and could give us a mechanism to prevent name clashes. And try to make full use of existing classes and structure to keep our code try (Improve utilization rate and reduce coupling degree).

For ground (Valley, Cemetery, Bonfire, and Vendor):

I add new terrains (**Valley** and **Cemetery**) that inherit **Ground abstract class**, they will get all features that Ground has (display character, location, and so on) and implement themselves special capabilities on themselves class. I also consider the **Bonfire, Vendor** and **FireSquare** burned(created) by **Ember Form action** as kinds of ground.
Because, they have similar properties or features with kinds of grounds (For example, selling weapons on the vendor is similar to killing the player when the player falls to the valley). We can consider those features as a special capability. An actor can do some actions like trading with the vendor when the pre-condition is meeting, for example, Yhorm’s Great Machete can call **EmberFormAction** to generate **FireSquare.**

As you can see, Cemetery uses **RandomRate** class to get success rate, I put this feature to a class because the function of generating random rate is used many times.

For weapons:

I divided the weapons into two categories, one is an ax and the other is a sword. In the future, more weapons can be added, and the same function of the same type of weapon can be placed in an abstract category, which conforms to the principle of "keep code dry". For each category, there are two classes separately, these classes will responsible for their special skills including passive and active effects. All weapons will use **RandomRate** to get a random rate to satisfy the requirements and use **Ability enum** since they all have different abilities(skills). **StormRuler** and **YhormMachete** have two kinds of **status** (active or not), so, they will use the **Status enum**.

For Actors:

Player, Undead, **LordOfCinder,** and Skeleton inherit **Actors** abstract class, All has their abilities and status(for example, there are REST ability and HOSTILE_TO_NENEMY status for the player) any special skills or properties(like resurgence ability for skeleton). All actors will use different weapons with the requirements of “**YhormMachete** can only be held by **LordOfCinder**”, “**StormRuler** can only be used by player” and other enemies will use a random weapon. So, I build an association relationship between actors and the **Weapon** interface, all actors could use the **Weapon** interface to get a kind of weapon base on requirements.

For EstusItem, PortableItem, and SoulsToken:

EstusItem cannot be abandoned, but it should belong to a kind of **Item**, so, I consider it as another single class instead of **PortableItem**, and **Player** has an association with it since the player could hold it. Moreover, **SoulsToken** is a kind of special item so it shouldn’t be included in those two classes. They inherit **Item** abstract class can be picked up by **PickupItemAction** class and can use all codes that an item needs.

For Actions：

The existing action classes are not shown in the class diagram.

I put all the action classes in the same folder, they all inherit the Action abstract, so that you can inherit all the functions of the **Action** and the impact of the related classes on the **Action**. But their targets are different (In the diagram, the purple line is connected to the targets). In addition, they all have an association relationship with those abstract classes, rather than directly with a specific class. This is convenient for us to modify the function in the future, for example, I want the action's target to become another one with the same abstract class parent.

ChargeWeaponAction is to charge LordOfCinder's weapons, and its target should be **YhormMachete**, therefore, the action has an association relationship with **WeaponItem**. The targets of **EmberFormAction**, **SpinAttackAction,** and **WindSlashAction** are actors, but these actions need to obtain the relative weapon first since these actions are weapon skills. When actors use them to attack other characters, they must understand which weapon released this skill.

I also added **RestAction**, this class is responsible for the actions that the player completes when using the rest skill in Bonfire. It has an association relationship with both Item and Actor abstract class because this class needs to refill the Player's health/hit points to the maximum, refill Estus Flask to maximum charges, and reset some stuff of enemies. So, we can see that both **Player** and those enemies implement the **Resettable** interface. The

RestAction class will use **ResetManager** to manage the objects that need to be reset with the help of the **Resettable** interface.

The **ResetAction** class is mainly responsible for actions that happened on the soft-reset process, and its target is those classes that can be reset (classes that implement the Resettable interface), so it has an association relationship with the Resettable class. When performing soft-reset, the reset process includes **RestAction**, so **ResetAction** and **RestAction** have a dependency relationship. **ResetAction** still need to use **ResetManager** to get objects that need to be reset.

DealAction is mainly responsible for the transaction between Vendor and Player, inheriting **Action** abstract class because it belongs to the action of the player, and the restriction is that player can only trade with Vendor, so I directly established an association relationship with **Vendor** class and **Player** class instead of with abstract classes(Actors/Ground).

For Interface package (some features and classes have shown above):

By observing the code given, I found that the **FollowBehaviour** class that implements the **Behaviour** interface is responsible for completing the enemy's actions to follow the player. This interface is mainly responsible for the behavior of each round of the enemy. In addition to the original several classes that implement the **Behaviour** interface, two more classes are added to implement this interface, they are **GenerateUndeadBehaviour** and **RecoveryBodyBehaviour**. They are responsible for Generate undead at a random rate and recovery corpses. All classes will play a role on every round turns like what the **FollowBehaviour** does. They have an association to **Actor** abstract class, so, these classes can be used by many actors and all enemies could have an association relationship with the

Behaviour interface. The design would improve utilization and reducing coupling, increasing scalability.

Player and **SoulsToken** class implement the **Soul** interface since both are suitable with the functions that Soul can do. For the player, it can transfer the Player's souls to another Soul's instance which could be **SoulsToken**, and add souls when the player defeats enemies and subtract after trading.