This UML class diagram mainly revolves on showcasing the reset system of the game using items, a unique ground, and the player. The goal of this design is to mainly implement a reset with specific effects that is dependent on the way a reset of the game is to be called while proving modularity, extensibility and maintainability.

Starting with FlaskOfCrimsonTears, it is a starting consumable item of the player that will never be dropped. Due to it being an item, it will extend the Item abstract class to inherit all the attributes of an item. As it is also a consumable, it will also have a dependency to the ConsumeAction class which extends the Action abstract class, to have all the attributes an action should have.

Next we have the SiteOfLostGrace. This is a unique ground and as such will extend the Ground class to have all the attributes of a ground. The player is also allowed to rest on this and with that, the SiteOfLostGrace and Player classes will be having a dependency on RestAction which extends the Action abstract class.

Now onto the game reset. First off, as all enemies will despawn and the player's hit points along with the number of consumptions of the FlaskOfCrimsonTears will be set to its maximum value again when a reset occurs, the FlaskOfCrimsonTears, and all the enemies and Player class which extend the abstract class Actors due to each of them being an actor, will implement the Resettable interface. Having all the enemies and the Player class extend Actors also follows the Don't Repeat Yourself (DRY) Principle as certain attributes will be present in every actor of the game. The reason Resettable is an interface is because not everything in the game will be influenced by a game reset. A ResetManager class was also created that has an association to the Resettable interface with the purpose to manage any operation involving the reset. The ResetManager also associates itself to ensure that there is only one instance of it because changes will be made to any classes that will be influenced by a reset which leads to the need of only one reset action. Having the Resettable interface also helps in the reduction of the amount of dependencies between both ResetManager and Resettable entities. By doing so the design also adheres to the Dependency Inversion Principle and Open-Closed Principle since the we do not need to fix the ResetManager to accommodate new entities that are resettable while putting a layer of abstraction between the ResetManager and the ressettable entities.

Next, a reset will occur when the player "dies". Hence, the Player class will have a dependency on the PlayerDeathAction class which extends the DeathAction class that extends the Action class due to it being an action itself. As the death of the player causes a reset, the PlayerDeathAction will also have a dependency on the ResetManager. The reason for having a PlayerDeathAction class instead of just a DeathAction class is because the effects of a player's death is different from when an enemy is killed. This will adhere to the Single Responsibility Principle to avoid the DeathAction class being a god class. Besides that, when a player rests at the

SiteOfLostGrace, a reset occurs and because of that the RestAction class will have a dependency with the ResetManager class.

Now for the runes. A rune is an item and as such extends the abstract class Item to have the attributes an item should be having. A RuneManager class was also created that has an association to the Runes class. The purpose of the RuneManager is to manage any operation involving runes, such as increasing or decreasing the amount of runes. This is due to the fact that the system is in charge of managing any rune operations. The Rune Manager also associates itself to ensure that there is only one instance of it.

When the player dies, their runes will be left on the ground and to do so, the Player class will have a dependency on the DropItemAction class which extends DropAction class that extends the abstract class Action as it itself is an action. The reason for also having a dependency to DropItemAction is because the rune will be dropped and runes are an item.

When the player stands on the dropped runes, the player is given the option to retrieve the runes hence the Player class also having a dependency to the PickUpItemAction class which extends PickUpAction class that also extends the abstract class Action due to being an action as well. The reason for also having a dependency to PickUpItemAction is because the rune will be picked up and runes are an item.

When the player drops or retrieves runes, the total amount of runes the player will have either decreases or increases respectively. As such, both the DropItemAction and PickUpItemAction class have a dependency to the RuneManager class.

If the player dies once more before retrieving the dropped runes, the runes will despawn hence the Runes class implementing the Resettable interface as a reset will have an effect on the state of the runes.

This UML diagram provides extensibility, modularity, and reusability through polymorphism and inheritance as stated above. For example, rather than creating a class that is responsible for death of all the enemies and players, a PlayerDeathAction class was created that inherits the abstract class DeathAction attributes as the effects of a player's death is different from when an enemy is killed which achieves modularity. By using the parent class as the type, we can also allow extensibility through polymorphism by allowing the addition of new subclasses without modification on the code since polymorphism allows a code to perform a single action in different ways. Hence, the design adheres to the Open-Closed Principle, Dependency Inversion Principle, Do Not Repeat Yourself (DRY) Principle, and the Single Responsibility Principle.