**Design Rationale**

**All Enemies:**

For the abstract Enemy class, we decided to have it inherit Abstract class Actor. This would let it inherit hp stats attack stats which would be shared amongst Actor abstract class. Also, by making Enemy an abstract class as well, we follow the Dependency Inversion Principle, which allows easier implementation of new Enemies.

The alternative methods that we were considering was to have Enemy as an abstract class on its own or have it as an interface. The problem with the first method would be the repetition of code, especially with health and attack stats that would already be in Actor class. The second method would mean that every new Enemy would directly inherit from Actor class, which may create some repetitive code, especially for rune dropping, which will not be used in User class. This adheres to Interface Segregation, Open/Closed and Single Responsibility Principle by making new Enemy subtypes easier to extend and not depend on any unnecessary interfaces.

The addition of the EnemyType enum was to have an enemy type that would be assigned to each new Enemy subclass. An example would be SkeletalType for HeavySkeletalSwordsman and SkeletalBandit. This would allow enemies to recognise their own types and not attack each other. We considered using magic numbers and string comparisons as alternatives. The first method would be too abstract and not aid in understanding the code. Both String comparisons and magic numbers would be too repetitive and violate Open/Close principle as we would need an if else statement to compare between enemy types. By using an enum, we promote cleaner code, easier maintainability, and extensibility.

Additionally, to calculate the probability of attacks by enemies, the class RandomNumberGenerator with a static method is used. This follows the DRY principle because the implementation makes it so that the method is only defined once as well as making it available to other classes.

We decided to make an abstract weapons class where weapons could inherit attack stats and easily extend the addition of new weapons. We also made an IAOECapable interface to describe the behaviour of AOECapable things, such as weapons and enemies. This would allow it to override attack moves and damage surrounding enemies.

Likewise, the creation of IBonePile interface is used to describe the ability of the skeleton type characters to turn into a pile of bones. This way we can follow the Single Responsibility Principle in case of the creation of a new Enemy that would be Skeletal type but doesn't turn into a pile of bones.

**Grace and Reset:**

We made the FlaskOfCrimsonTears inherit from an abstract Item class, which would allow Player to interact with the item in set methods, rather than having to create a new method whenever a new item is created. It's usage will be implemented through a IConsumable interface which defines the usage of the Flask.

We used an interface that connects to GiantCrab, LoneWorld and HeavySkeletal Swordsman which will be used to set a method where each respective enemy is reset. The idea behind using an interface is so that not every entity be resettable.

For SiteOfLostGrace, we had it connect to an abstract class Ground, which will be used for other ground types such as RuneCollectable, for when there are runes on the Ground. SiteOfLostGrace and RuneCollectable will have dependecy relationships with the player to signify if the player will collect runes or reset the game.

**Runes:**

To implement runes in the system, creating an entire class for it would have been wasteful. As runes only function as a currency, a rune as an object would not have any attributes to hold and if carried by an actor, each rune carried would be its own object. Therefore, runes are represented as an attribute of the actor as all objects that extend it can carry runes. Additionally, to calculate the random number of runes possessed by enemies, the RandomNumberGenerator class is used which follows the DRY principle as the class is reused to perform a function which allows the code to be written cleaner with no redundancy.

**Trader:**

The trader and player class are implemented by inheriting from the actor class and both follow the Dependency Inversion Principle as these concrete classes depend on the abstract class weapons. This allows the player and trader to have only one array for holding various objects that extend the weapon class which also adheres to the Open-closed principle as added a new weapon class will not require a new array for the actors to store the weapon. Additionally, the buying and selling price of every weapon is contained in their own enums. This is an ideal implementation as these values are constant for their respective classes and it removes the need for magic numbers.

**Environments:**

To implement the environments, each environment class inherits from the ground class and follow the Dependency Inversion Principle as these concrete classes depend on the abstract class enemy. This makes it so that each environment can spawn enemies of different types if the code is ever extended to include new enemies that have the same spawn environments which also fulfills the Open-closed principle as the new arrays do not have to be created. This implementation however does allow enemies of different types to be spawned in locations they do not belong, which can be solved by validation through checking enemy type.