REQ 1 Design Rationale

The diagram represents the first requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring.
We have three design goals for this requirement: implementing environment, implementing enemies, and implementing weapons.

For the environment; Graveyard, GustOfWind, and PuddleOfWater are all extended from the Ground abstract class. This is due to all three of the environment classes having the same common attributes and methods, so we abstract these classes to avoid code repetition (DRY).

We created a new abstract class called Enemies so that all the enemy classes (HeavySkeletalSwordsman, LoneWolf, GiantCrab) can extend from it and do not have code repetition since they would all have common attributes and methods (DRY). All the enemy class will require to act on the behaviour classes. And by using the Behaviour Interface, we can have the Enemy class act on the Behaviour interface rather than depending on each behaviour classes, this way we can easily modify it for future use (DIP).

All the behaviour classes are implemented from the Behaviour interface, this is due to class like AttackBehaviour implementing both the Behaviour interface and also the Weapon interface, if both of them were to be classes this wouldn't be able to work. And new set of operations could be easily added to the system by adding more interfaces since each of the current interfaces is separated by its own purpose (Interface Segregation Principle).

Some enemies can decide to either attack a single target or multiple at once, so in order to not let AttackAction having both responsibility, a new class is created to handle attacks on multiple enemies on an area, and let AttackAction to strictly handle single target attacks. (SRP)

Grossmesser extended the WeaponItem abstract class. Since all weapons share some common attributes and methods, it is logical to extend Grossmesser from WeaponItem abstract class so that Grossmesser have all the attributes and methods that a weapon should have without code repetition (DRY).

We then also package all the classes and interfaces into their respective package for organization, modularity, and encapsulation.