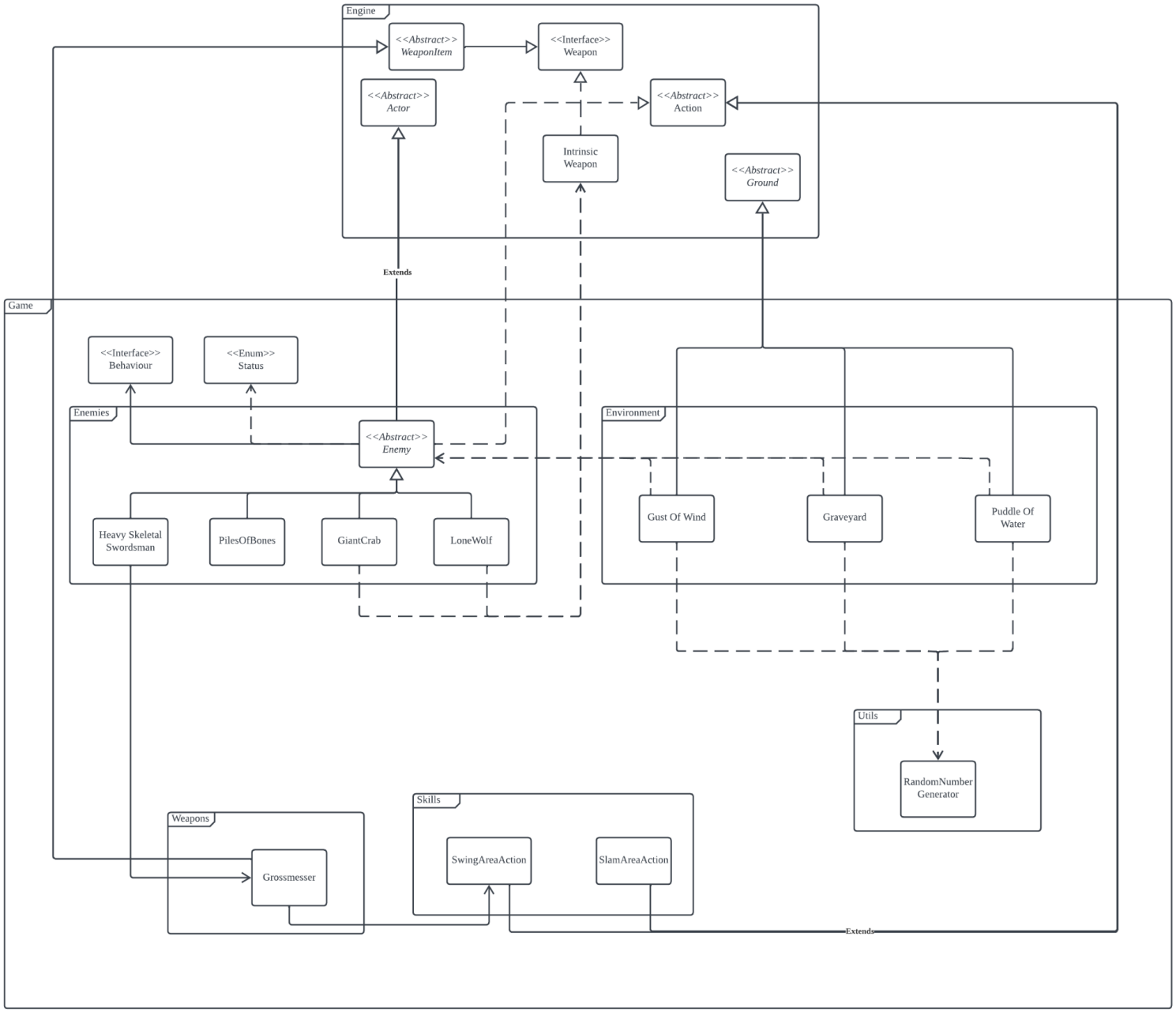
REQ1



The UML diagram represents the basic characters and grounds for the Elden Ring Rogue-like game.
In this UML diagram there's mainly 3 concrete Ground classes and 3 concrete Actor classes.

-Design Goal: Able to spawn different types of enemies from different grounds with specific probability.
All 3 types of grounds are extended from the abstract Ground class since they have common methods.For enemies , we have created an abstract Enemy class because all enemies will have similar methods. These two design decisions are made by considering the DRY

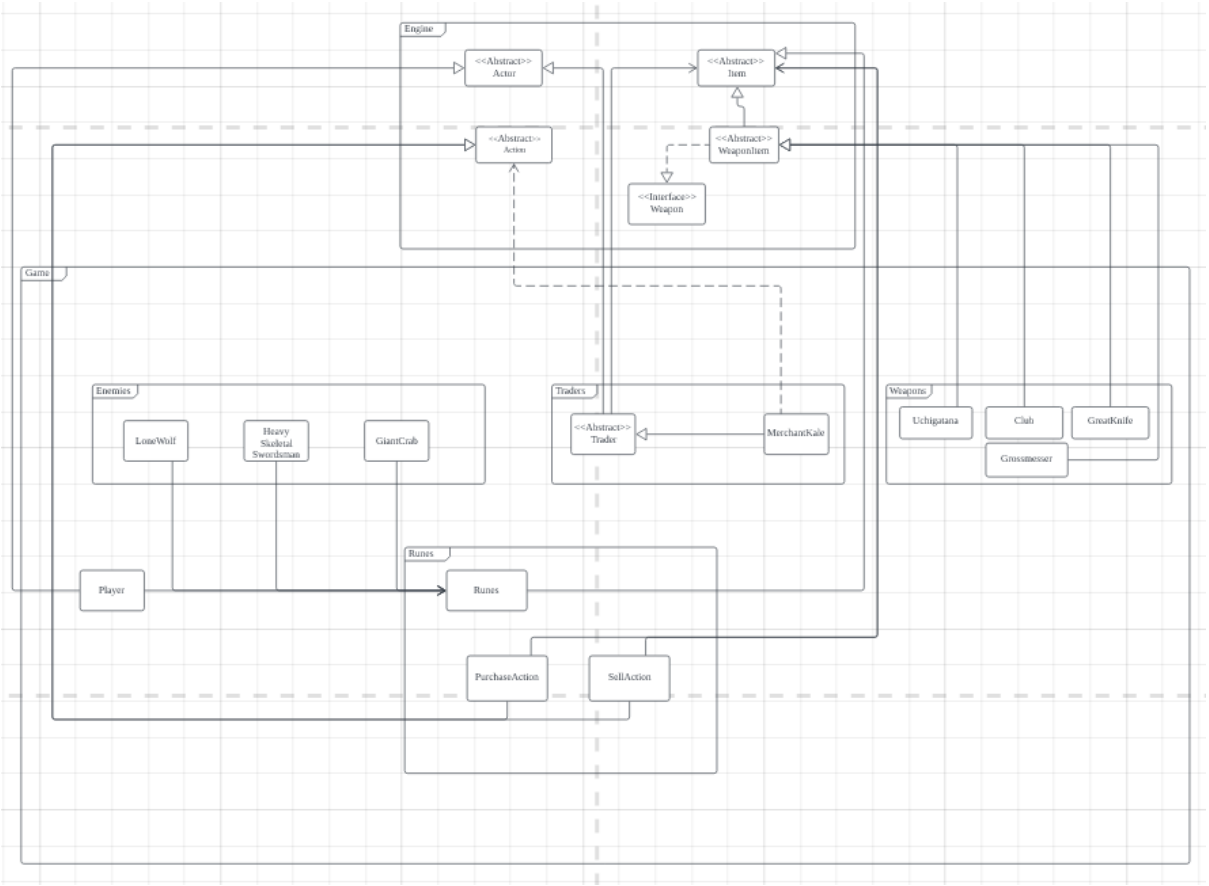
principles by avoiding repetition. To compare and contrast, let's assume we take the alternatives which is to implement all of the grounds without having dependency on abstraction. Let's say we have addItem() method for all the ground types, which is meant to add Item to the specific ground, then we might have repetition of implementation of addItem() method.

Instead of having a god class that is responsible to create different types of grounds and spawning enemies respectively, we made a separate ground class which is only responsible to spawn specific types of enemy.This way we achieved Single Responsibility Principle(SRP). In other words, instead of having a big class named Enviroment, which is responsible for creating grounds with different names ,then having a method of checking the type of the ground to spawn different types of enemy, we had a separate class for grounds. But, the trade off is that we will be having more dependencies since we have more class.

-Design Goal: Each enemy will behave differently under different circumstances. For example, Enemy should follow the Player if the player is one block away from the enemy. Hostie creatures should wander around the map.
Other than that, we have created different interfaces, which is known as "Behaviour". This is meant to be used by the NPC for their action, in other words how they behave. We have created a small size interface. For example, instead of having one big interface named 'NpcBehaviour ", which might have a lot of different methods since NPC can do many things, we had a small interface like "FollowBehaviour". So, when a NPC will follow the Player they will implement this interface, in this case there's a much higher chance that the NPC class will fully utilise the interface without having a method not implemented. In this case we achieved the Interface Segregation Principle.
As, we can see in the UML, the different grounds all depend on the <<Abstract>> Enemy class, in this case we achieve Liskov Substitution Principle(LSP), if we take any enemy class e.g. Lone Wolf, it will still work as expected.

-Design Goal: Enemy with same species cannot attack each other
With this we have implemented it using <<Enum>>, each enemy should have a <<Enum>> type, specifying which type they are. So before performing an attack action on a certain actor, they will check their <<enum>> type, if it's of the same type , then they won't attack. With this implementation we have achieved the Open-Closed Principle(OCP) .Let's say we need to have even more different types of enemy, this implementation will still work.

REQ2



The diagram represents the interactions between MerchantKale and the players. It also includes the relationships between Runes and all Actors that are associated with the Runes class. In the game, players are able to buy/sell items when interacting with merchants/traders.

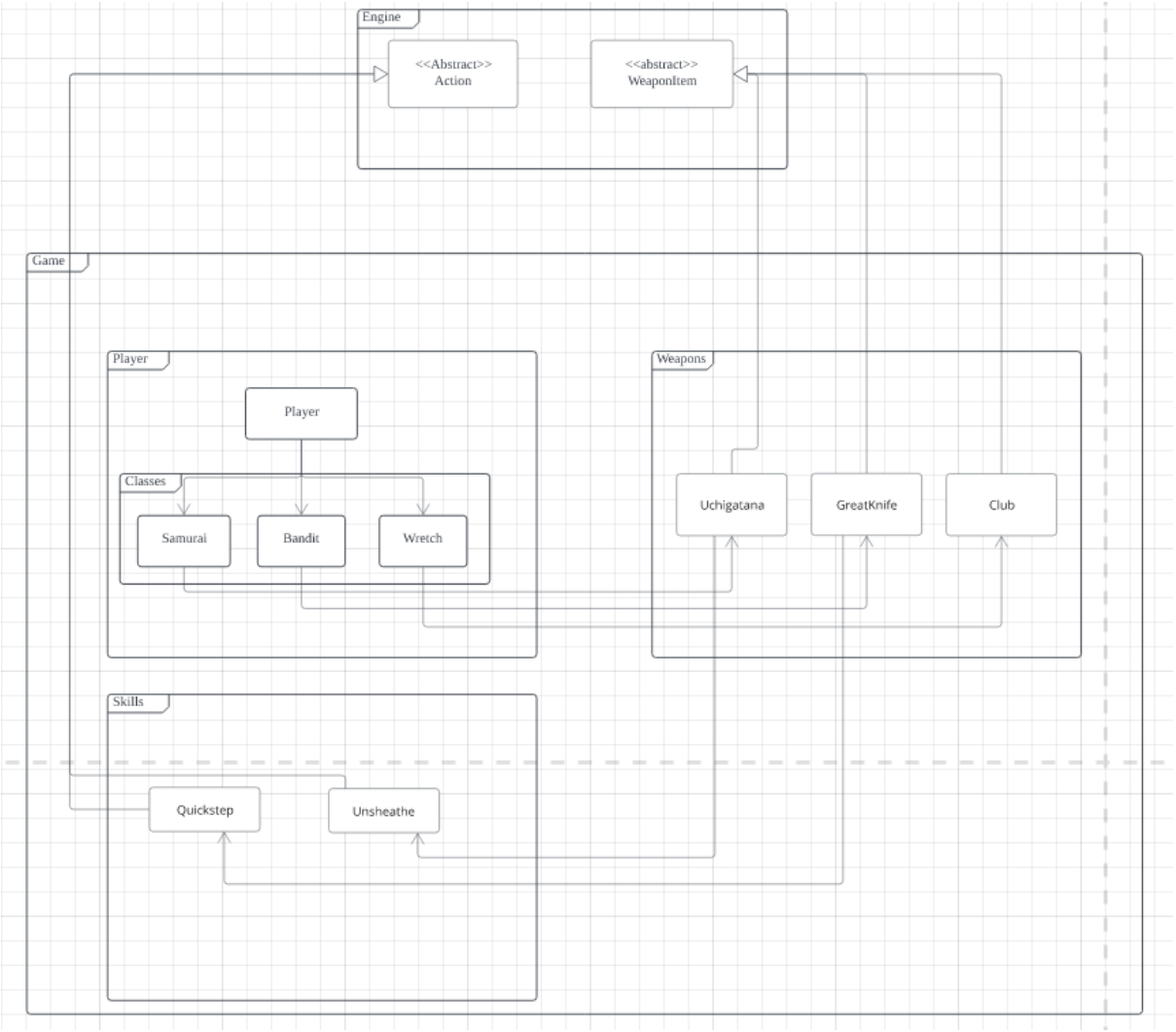
In our design, the MerchantKale class is a subclass of an abstract Trader class. The aforementioned abstract class was created in the event that more merchants with similar attributes and methods to MerchantKale are added to the game, this is to avoid repetitions of similar code that only merchants have (DRY).

Runes are represented as the Runes class which is a child class of the abstract Item class, this is to allow actions to be performed on Runes. All players and enemies will have an association with Runes as both classes will store an instance of Runes, this allows us to track the number of Runes that a player/enemy has as Runes will have an integer attribute that represents the value of that Runes item.

PurchaseAction and SellAction are child classes of Action. In this design, they allow interactions between the player and traders, giving players options to buy and sell items. The reason for separating transactions into 2 separate classes is to prevent overcomplication of the code. Both classes have an association with the Item class, allowing transactions to also involve items that are not weapons.

Traders have an association with the Item class. This accounts for future requirements that ask for traders to be able to buy/sell items that are not weapons, preventing any errors from occurring. This follows the Open-Closed principle.

REQ3



The diagram covers the scenarios involving interactions between the player and the following: FlaskOfCrimsonTears, SiteOfLostGrace, ResetManager, and Runes. All of which are represented as their own class.

FlaskOfCrimsonTears is a child class of Item. As this item is currently exclusive to the Player, Player will have an association with FlaskOfCrimsonTears, allowing the player to interact with the item on each turn.

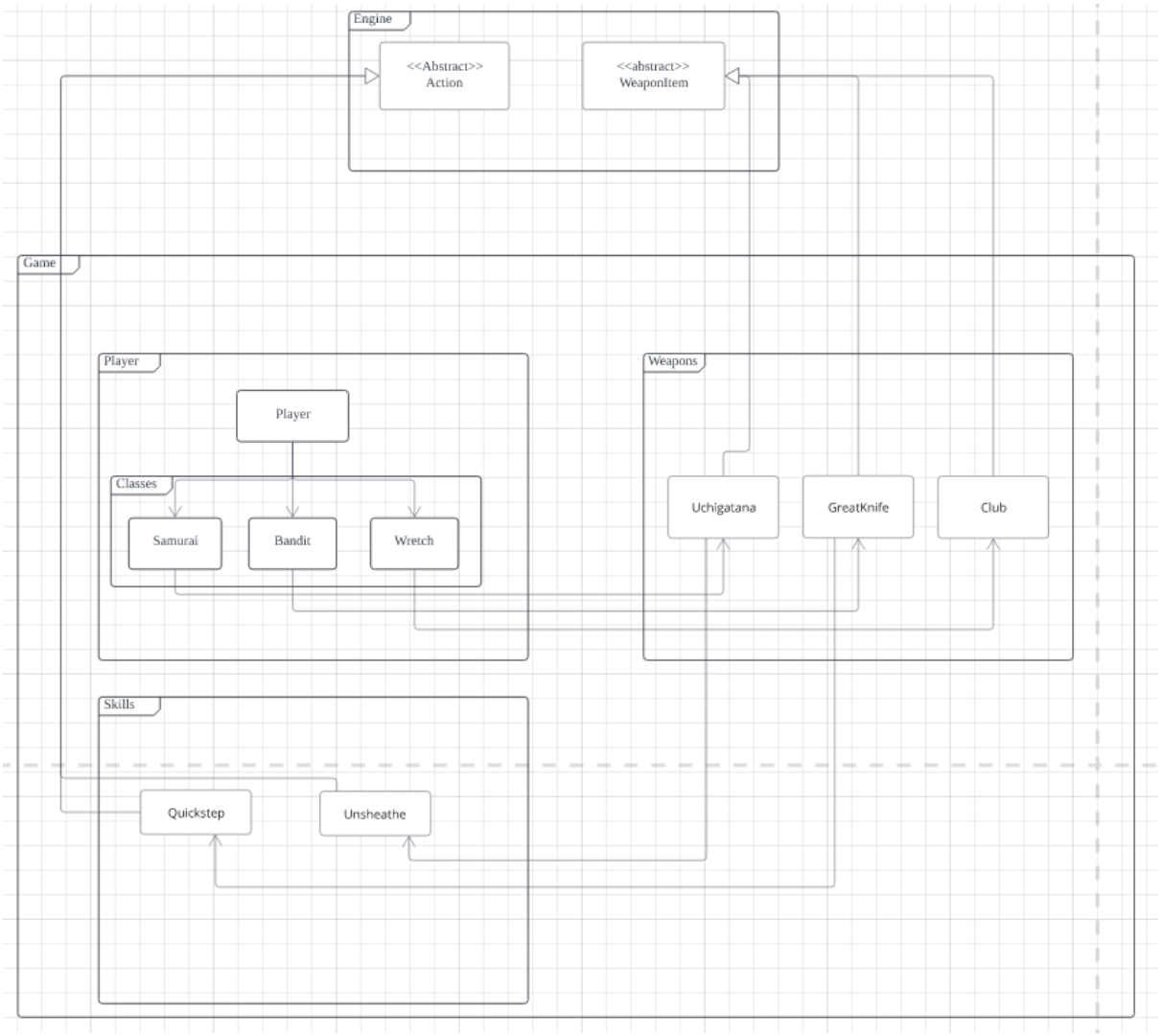
Currently the Player can only interact with the flask through ConsumeItem, which is a child class of Action. FlaskOfCrimsonTears is associated with ConsumeItem as the latter is an Action which can be performed by the player on the former.

SiteOfLostGrace is a child class of Ground class. It has an association relationship with RestAction, which in turn is associated with ResetManager. The ResetManager class is associated with the Resettable interface, allowing the manager to reset all classes that implement the interface. Using this design, adding new resettable objects will not result in changes in the ResetManager class. This follows the Open-Closed principle.

DeathAction is associated with ResetManager so that in the event of the player's "death", the game will reset. DropRunes and RetrieveRunes, both of which are child classes of Action, are associated with Runes. DropRunes and RetrieveRunes are not child classes of DropAction/PickUpAction as the player-interactions between runes and items are slightly different. For example the string representation when picking up an item is different from picking up a Runes item.

In the event that a player dies without retrieving their dropped Runes, their Runes will despawn. This is achieved through DeathAction having a dependency relationship with DespawnRunes.

REQ4



Design Goal: A player will have an option to choose 1 out of 3 starting classes and every class will have its starting weapon. Two of the starting weapons can use skill.

The diagram represents an object-oriented system for starting class of player, weapon of the starting class and the skill of the weapons.

All 3 starting classes are dependencies of Player class. It is better than using inheritance because when using inheritance, changes made to the superclass may affect the behaviour of its subclasses, it may cause maintenance problems and code complexity.In addition, child classes will have to fulfil all contacts defined by the parent class. All methods in the parent class will be exposed to child classes, which could make the child classes bloated with things that it doesn't need. It uses Open-Closed Principle(OCP) since we can have more starting classes if we want and it will not break the player class.

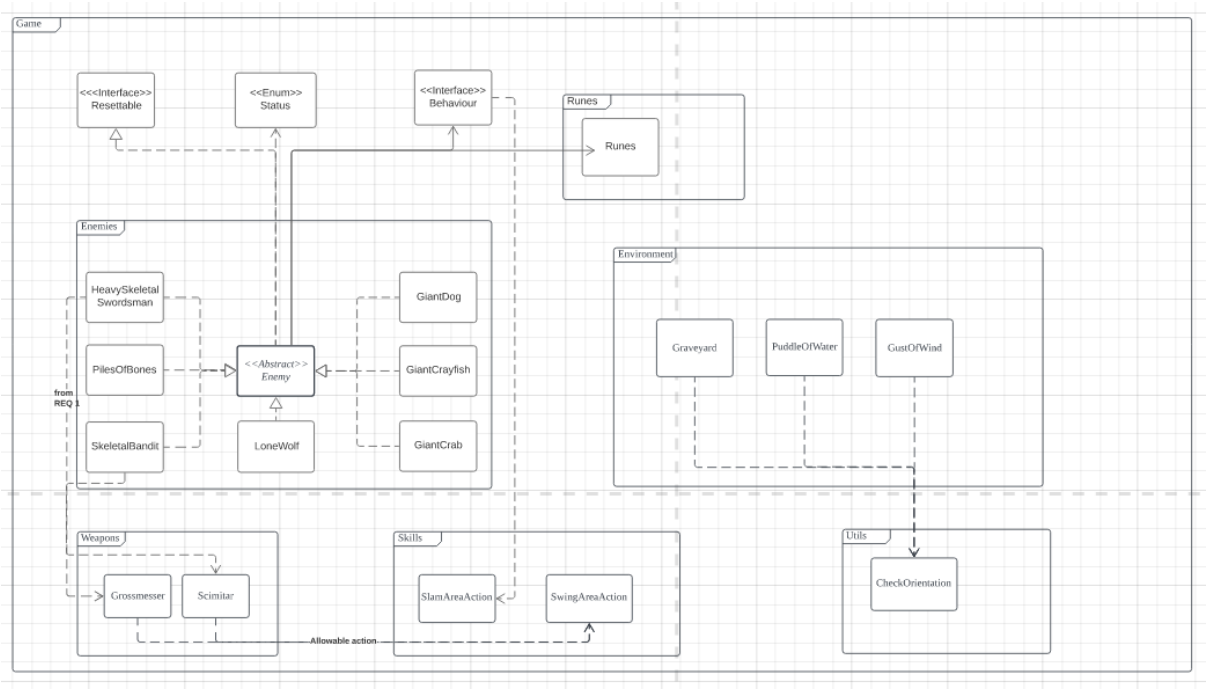
All 3 weapons extended the abstract WeaponItem class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY).

All 3 weapons classes will have a dependency on the respective starting class because those are the starting weapons of those classes so making them attributes of those classes is necessary .

Weapons that are stated to have a skill ability have a dependency on their respective skills to indicate that said skill is an attribute of the weapon.

All 2 skill classes are extended from the Action class. The reason for not creating an independent skill action class and letting the skill class have dependency on it is because it will overcomplicate the code. It also achieved Single Responsibility Principle(SRP) because each skill class will only do its own skill.

REQ5



The UML diagram represents advanced creature and ground implementation.

-Design Goal: Split the starting map into the Left (West) and Right (East) half, then spawn enemies respectively.
In other words, inside the ground class we need to identify the x coordinate of that specific ground. Instead of having to repeatedly check for the coordinate in every ground class that is created using , if x<.. return someEnemy() else return otherEnemy(). We create a class call CheckOrientation class, which provides a method to check coordinates. So inside the ground class we just need to call a static method inside the CheckOrientation class to help us check the coordinates. This case we achieve Don't Repeat Yourself(DRY) , we don't need to write the same method in every single class instead we just call the static method.Other than that , the Open-Closed Principle(OCP) is also achieve, in the sense that if we not only need to check for x coordinates ,but y as well. Since we are calling a single method from the CheckOrientation class, we just need to modify the method inside that class, instead going to each class of ground to modify.

As mentioned in the requirement, Scimitar and Grossmesser have similar skills. Hence , both Scimitar and Grossmesser return the same action , instead of writing another action class for Scimitar abilities. This implementation achieved (DRY)