Design Rationale

FIT2099 Lab 4 Group 2

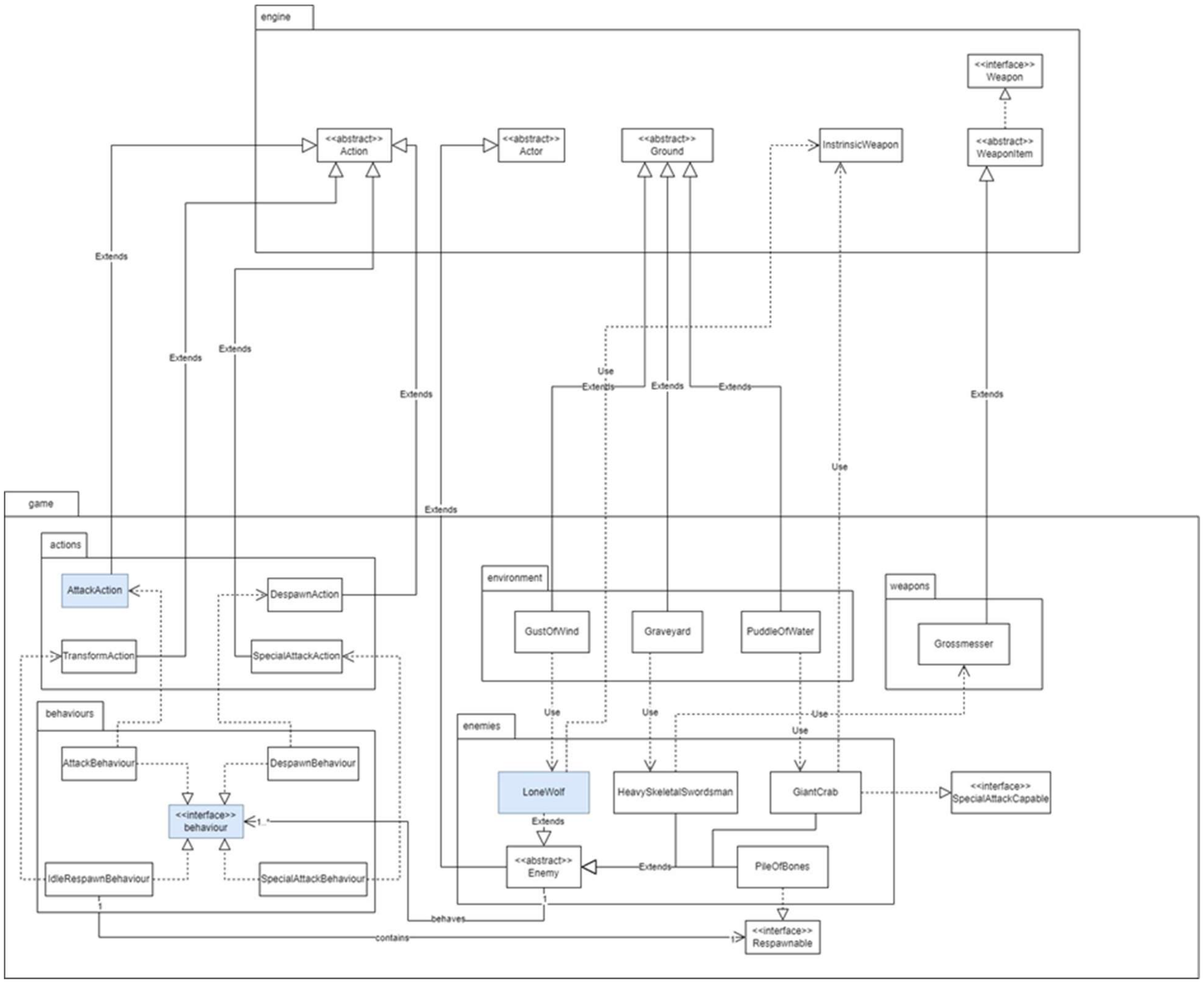
Members:

Keith Ong Guo Er	32287089
Sam Zachary Chee	32805195
Yap Wing Joon	31862527

* Note that classes that are in bold represents new classes that we have created for this requirement

Requirement 1 - Environment and Enemies

Class Diagram:



Design Rationale:

Firstly, we created three new classes which are **GustOfWind**, **Graveyard** and **PuddleOfWater** in the environment package which represents the environments where enemies are spawned. These three classes would all extend the Ground abstract class as they share common attributes and methods. Hence, it is better for these classes to inherit their methods from the parent class Ground to adhere with DRY principle to avoid repetitions.

We then created two other classes representing enemies that can spawn based on their corresponding environment apart from LoneWolf class which are **HeavySkeletalSwordsman**, **GiantCrab** and **PileOfBones** which all extends another abstract class **Enemy** that we have created as they all share the same attributes and methods to avoid repetitions according to DRY principle. The **Enemy** abstract class extends the abstract Actor class so that it inherits all methods from the Actor class as well. The purpose of creating an **Enemy** abstract class is so that we only need to add methods needed for all enemy characters to the abstract class which corresponds to the Open-closed Principle. Hence, the child classes would be able to inherit the methods and be implemented. Alternatively, we could just extend all enemy characters to the abstract class Actor in the engine package without creating an **Enemy** abstract class. However, doing so would add methods relevant only to enemies to other actors as well such as Player which is incorrect based on the requirements and would also lead to tight coupling. Hence, we decided on creating an **Enemy** abstract class as we believe that it would be easier to maintain their methods and attributes in the long run.

There are three enemy character classes, LoneWolf, **HeavySkeletalSwordsman** and **GiantCrab** which spawn in the grounds **GustOfWind**, **Graveyard** and **PuddleOfWate**r respectively. Therefore, each child ground class would have their respective single dependencies to their relevant enemies.

We also created other interfaces such as **Respawnable** and **SpecialAttackCapable** apart from the interface, Behaviour which was already provided in the base code. With that, we can avoid multi-level inheritance as **PileOfBones** and **GiantCrab** would only need to implement methods corresponding to their interfaces shown in the diagram above. Hence, **PileOfBones** would be able to respawn and **GiantCrab** now has the capability to perform a special attack. To justify our approach, we would now be able to add more methods and operations to our game by adding them as interfaces as each has its sole purpose which corresponds to Interface Segregation Principle.

We have also added 4 other classes of behaviours, **AttackBehaviour**, **DespawnBehaviour**, **IdleRespawnBehaviour** and **SpecialAttackBehaviour** which all implements the interface Behaviour. By doing so we only need a single hashmap of type Behaviours to store all the behaviours instead of adding hashmaps for all behaviours. This adheres to the Dependency Inversion principle. These behaviours all describe actions that are available for the enemy to act on. For example, enemies have the option to attack. Since the AttackAction class was already provided, we created an **AttackBehaviour** class to link the action to that of the enemies through the getAction method in the behaviour interface. Hence, enemies can now have the behaviour to attack. The same goes for the other behaviours. So, each behaviour class would have a dependency to their related actions which led to the decision to create an action for each behaviour. Based on our diagram above, the action classes would be **AttackAction, DespawnAction, TransformAction and SpecialAttackAction** which is all depended by the behaviours respectively.

Apart from that, **IdleRespawnBehaviour** has an association to the interface **Respawnable** as it contains attributes of type **Respawnable** necessary for **PileOfBones** to respawn using the **TransformAction.**

The **Enemy** abstract class has an association with the Behaviour interface as each enemy would have one or many specific behaviours every turn. The class would initialise an attribute of type Behaviour as a hashmap to store the behaviours with keys of type integers to represent their priority. With this, we have removed multiple associations for **Enemy** abstract class on multiple classes.

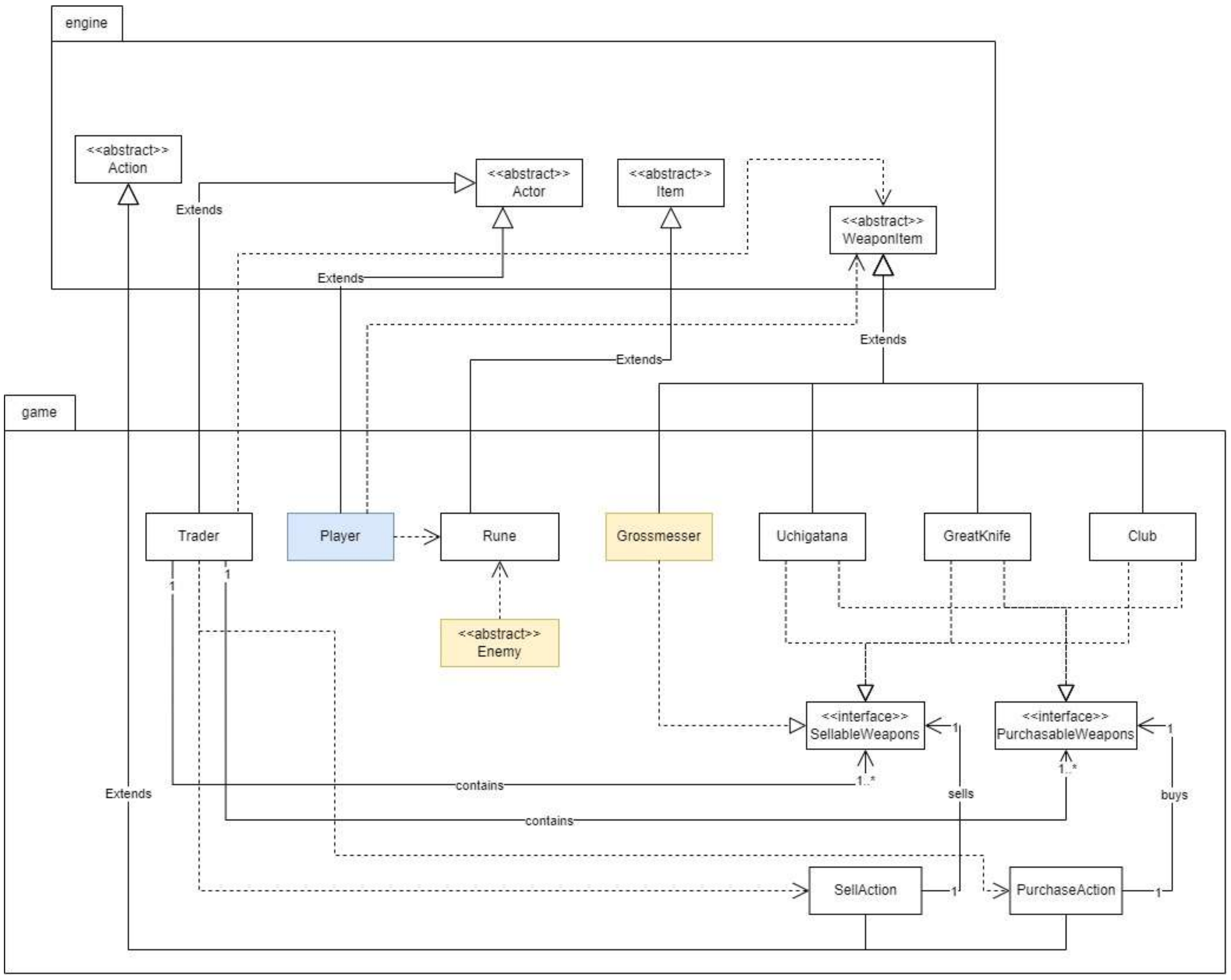
Lastly, we created a weapon class, **Grossmesser**, which extends the abstract class WeaponItem which implements the Weapon interface. By extending this class to the parent class WeaponItem, it inherits all the necessary methods to determine the characteristics of this weapon which adheres to the DRY principle once again when implementing its characteristics. This class is used to create an object of type **Grossmesser** which is equipped as a weapon by the **HeavySkeletalSwordsman.** Hence, **HeavySkeletalSwordsman** would have a dependency towards the **Grossmesser** class.

*Note that classes that are in bold represents new classes that we have created for this requirement

*Classes that are in blue are classes provided in the base code

Requirement 2 - Trader and Runes

Class Diagram:



Design Rationale:

For this requirement, we first created a class **Rune** which extends the abstract class Item so that it inherits all methods of the abstract class. With that, runes have characteristics of items and can be picked up. The purpose of this class is to create a **Rune** object which can be added to the inventories of players and enemies. So, the **Rune** class will be depended by **Player** class that extends the abstract class Actor which we was provided as well as the abstract class Enemy which was created in the previous requirement.

Next, we created a **Trader** class which extends Actor as well as other weapon classes **Uchigatana**, **GreatKnife** and **Club** which extends the abstract class WeaponItem**.** The relationship between traders and weapons will be established further throughout the rationale.

By extending the weapon classes to the abstract class WeaponItem like Grossmesser in our previous requirement, it adheres to the DRY principle which avoids multiple repetition as the weapons all inherits the methods characteristics of the parent class WeaponItem. Hence, this describes maintainability of code in the long run if more weapons are added. **Trader** and **Player** also depend on the abstract class WeaponItem. This way, we have removed multiple dependencies on Trader and Player to multiple weapon classes. We do not need to modify **Trader** and **Player** classes to add new functionalities to our code. This adheres to Open-closed principle. Now, players can carry these weapons in their inventory.

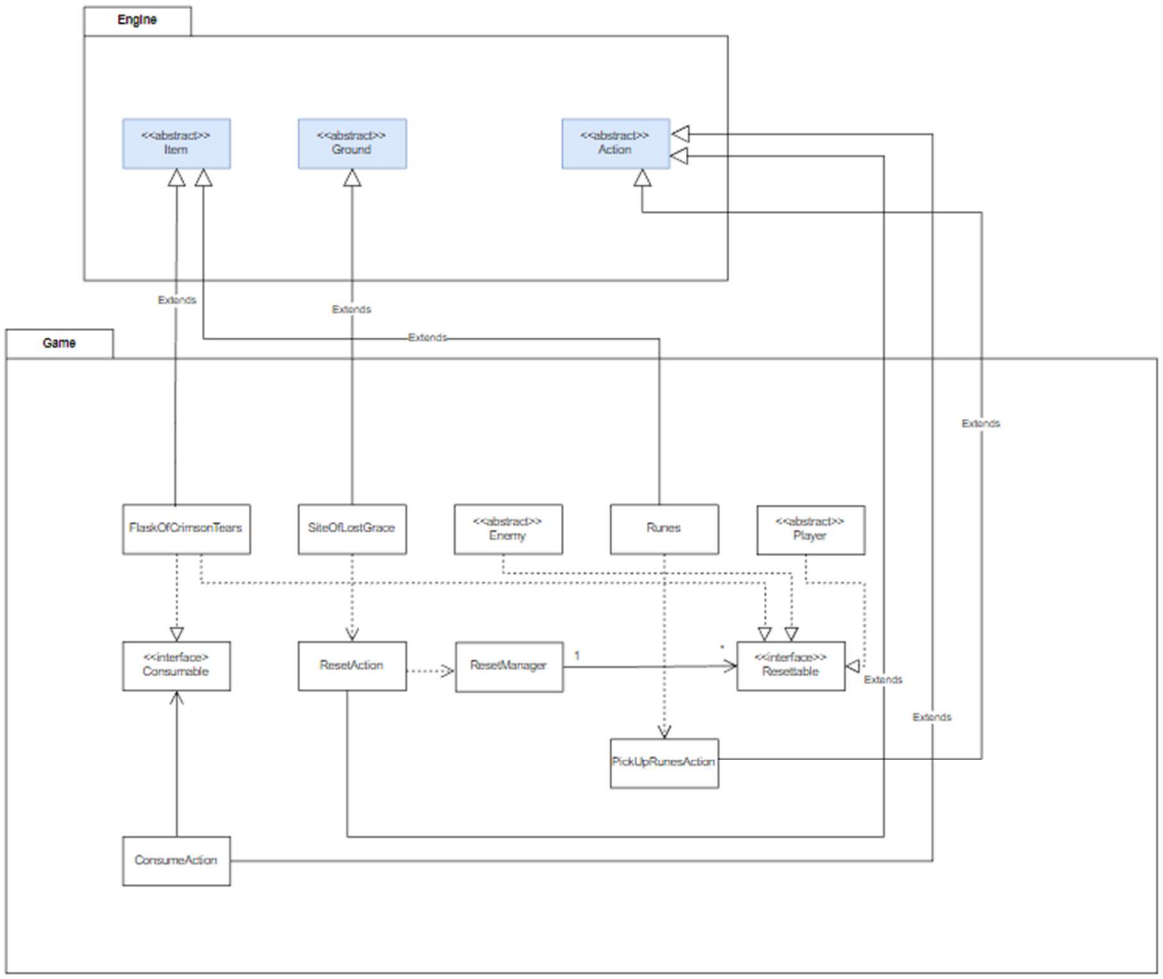
We then created two interfaces, **SellableWeapons** and **PurchasableWeapons** which are both implemented by all weapon classes except Grossmesser which only implements **SellableWeapons** as it can only be sold to the trader. Once again, by creating these interfaces, we are avoiding multi-level inheritance and since each interface has its own purpose. We have adhered with the Interface Segregation Principle.

We have also created two new action classes which are **SellAction** and **PurchaseAction.** Both classes extend the abstract Action class which means that they inherit all methods and characteristics of the parent class, Action. **SellAction** has an association to **SellableWeapons** where each sellable weapon has a sell action. The class would initialise an attribute of type SellableWeapons to execute the sell action so that players can sell weapons to the trader. The same goes for **PurchaseAction** which has an association to **PurchasableWeapons**. In contrast, there are more weapons that can be purchased by the player from the trader. The relationships between the actions and interfaces have removed multiple associations for both **SellAction** and **PurchaseAction** to multiple weapon classes.

Trader would also have associations to **SellableWeapons** and **PurchasableWeapons.** A trader can have one or many sellable and purchasable weapons. The class would initialise attributes of both interfaces in methods to determine which weapons are able to be sold and purchased. Trader would also depend on both **SellAction** and **PurchaseAction**. This is because Trader will then be capable of selling items to the players or purchasing items from the players.

Requirement 3 - Grace & Game Reset

Class Diagram:



Design Rationale:

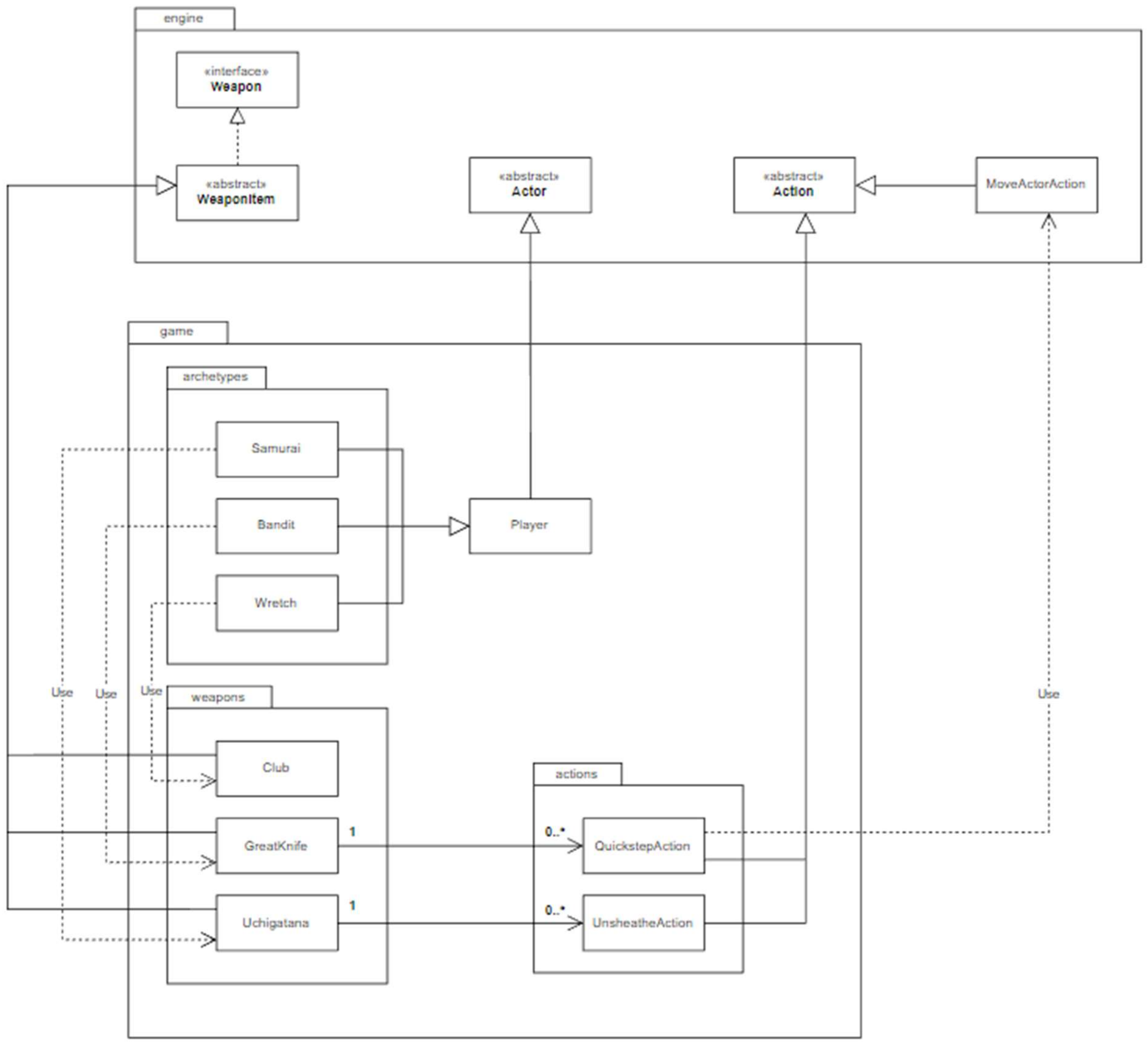
This diagram shows the a few items of Elden Ring, represented by concrete classes extending the Item abstract class, followed by the design of game reset. As a consumable item, Flask of Crimson Tear implements the Consumable interface which is associated to the ConsumeAction class. This implementation provides extendibility, in the case of adding new consumable items as the effect of each consumable item is implemented by item itself. The engine merely calls ConsumeAction whenever a consumable item is to be consumed by an Actor.

Another notable item is the Runes class, which contains an attribute called amount. The reason for this is that each Actor realistically can carry a lot of runes, which is unrealistic to track and be contained in an array list (i.e., 900 runes is equivalent to 900 instances of Runes class). Instead, the amount will keep track of the number of runes an actor should have. With such a property, the Runes class require a special pick-up action, which is implemented in PickUpRunesAction. The downside is that this approach would violate the Liskov-Substitution Principle as PickUpRunesAction override the Runes original getPickUpAction method. Consequently, this means substituting Item abstract class with Runes class would break every other subclass of Item class.

Game reset involves the Player resting at The First Step which extends Ground abstract class. During this, ResetAction class is called which creates and runs an instance of Reset Manager. Reset Manager invokes the reset method of all classes that implements the Resettable interface. Each resettable resets differently based on their own implementation. With regards to the DRY principle, Site of Lost Grace can be made an abstract class with future site of lost graces beyond the First Step extending it.

Requirement 4 - Classes (Combat Archetypes)

Class Diagram:



Design Rationale:

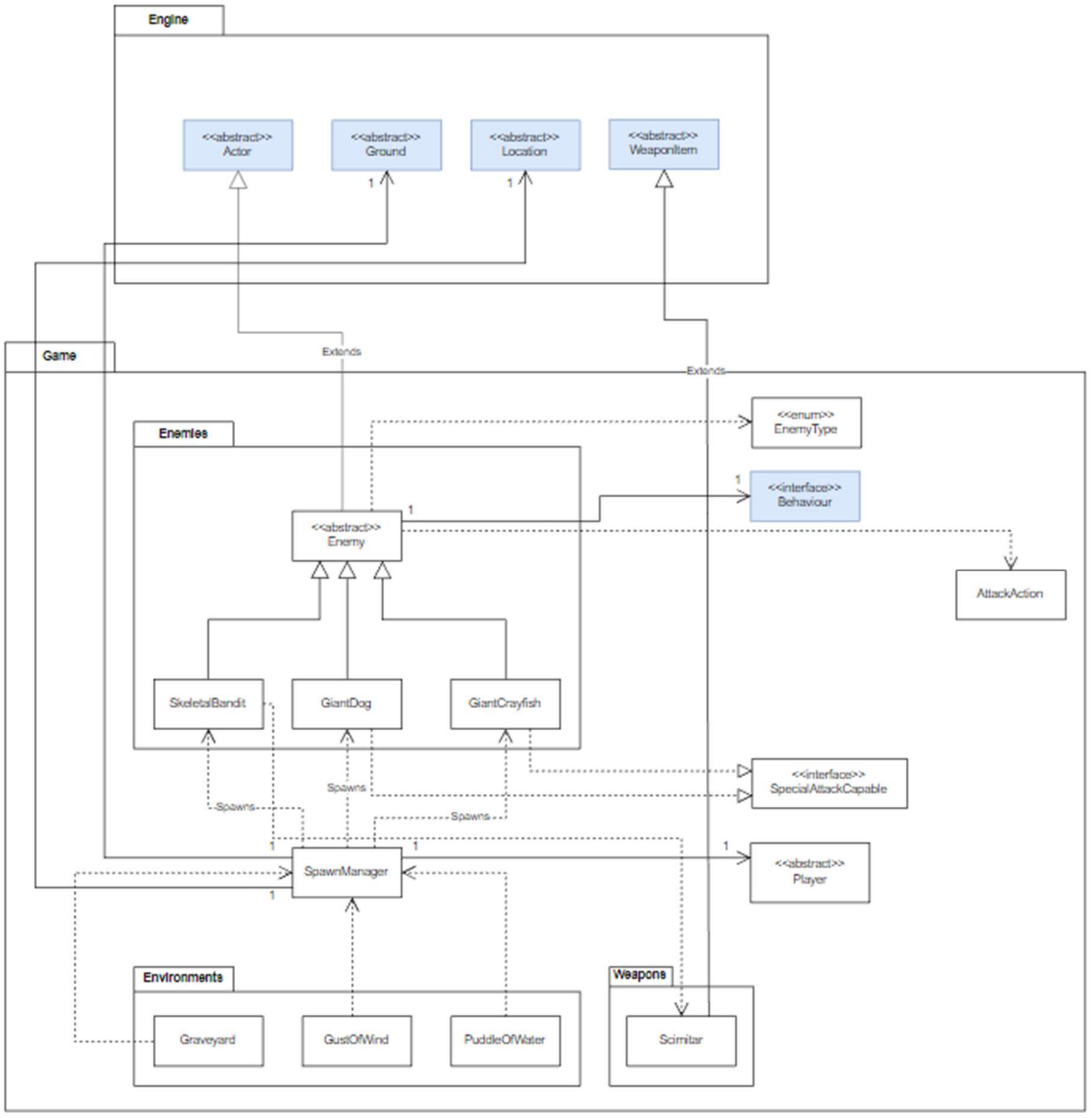
For this requirement, three new classes **Samurai, Bandit** and **Wretch** are created in the archetype package which are combat archetypes/ classes the player can choose from. These three classes extend the Player class which extends the Actor abstract class. They extend the Player class instead of the Actor class to follow the DRY principle to avoid repetitions when adding items such as the starting runes to the player's inventory (only need to be implemented in Player and not in any of the three other archetypes).

Next, we establish a dependency relationship between the **Samurai, Bandit and Wretch** classes with the Uchigatana, Great Knife and Club classes (implemented in Requirement 2) respectively in the weapons package. These weapons are added to the player's inventory at the start of the game based on the following relationships. Since these three weapons extend the WeaponItem abstract class which implements the Weapon interface they inherit the methods of both classes.

Lastly, we create the **Unsheathe** class as the unique skill for the Uchigatana and **Quickstep** for the Great Knife which extend the Action abstract class in the actions package. The weapons have an association relationship of 1 to 0 or many to their corresponding skills as one weapon can activate the skill 0 or more times. The **Quickstep** class is dependent on the MoveActorAction class as the skill allows the actor to move after attacking an enemy therefore requiring it. Alternatively, these skills could be implemented in the weapons itself however that would affect the reusability of the code if newly added weapons also had the same skill. Additionally, we wanted to make use of the Weapon interface which contains a method that allows the weapon to return a skill as an Action.

Requirement 5 - More Enemies

Class Diagram:



Design Rationale:

As REQ 5 is a vague combination of REQ 1 and 2, it uses a lot of similar classes and interfaces mentioned in those two requirements. This diagram features three new enemies represented by concrete classes extending Enemy abstract class and a new weapon represented by a concrete class extending WeaponItem abstract class and implementing relevant interfaces.

The three enemies’ concrete class is Skeleton Bandit, Giant Dog and Giant Crayfish. Giant Crayfish and Giant Dog implements the SpecialAttackCapable interface as they have the capability to slam enemies. Skeleton Bandit is nearly identical to Heavy Skeletal Swordsman, the only difference being it holds a different weapon, called Scimitar.

Scimitar is a sellable and purchasable weapon hence it implements both PurchaseableWeapons and SellableWeapons interfaces.