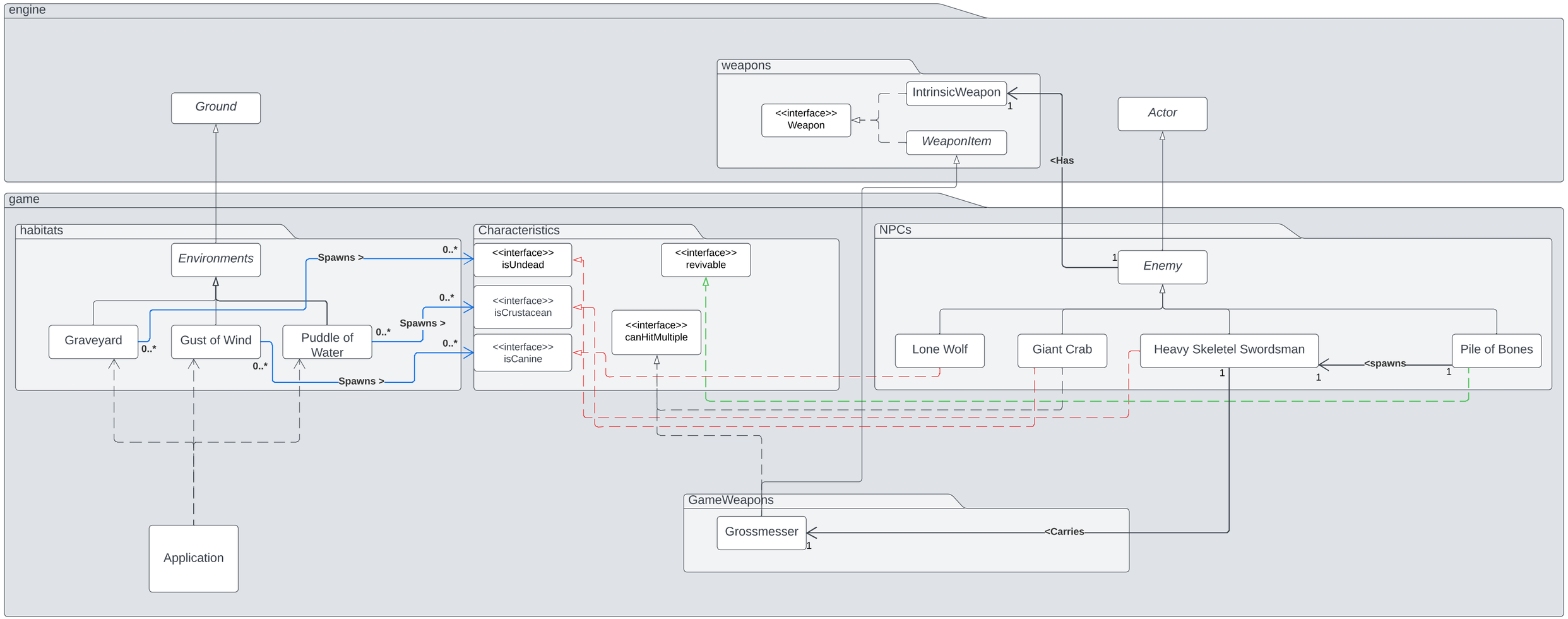


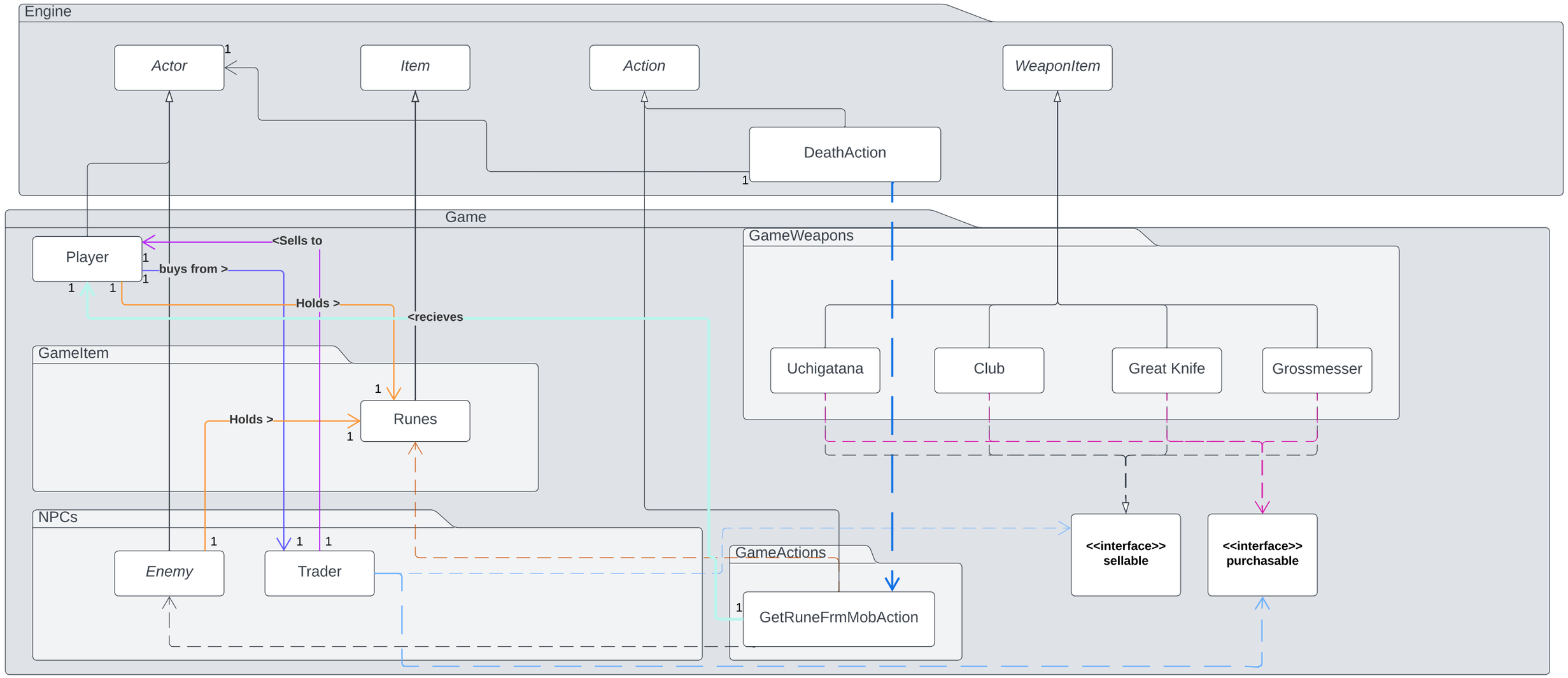
The diagram represents an object oriented approach for a system of enemies and environments in our game.

The two abstract classes environment and enemy extend parent classes from engine (ground and Actor), as well as also have their own concrete subclasses. Since these classes share some common attributes and methods this was employed to avoid repetitions.

The environments all assosciate with a type of enemy, done using interfaces e.g. isUndead hence enabling the code to be more extendeable as new enemies added in the future would be able to added through this interface therefore adhering to the open-close principles.

The interface revivable and canHitMultiple are interfaces are used to enable other classes in future to implement these methods adhering to open-close principle as well as abides by the interface segregation principle by having a singular and small purpose as well as being extensible for new interfaces to be added later on such as a canBeThrown interface being added. The implementation of the canHitMultiple as an interface also enables it to be inherited from different types of classes allowing for the code to be more extensible.

The pile of bones is a concrete class which inherits from revivable as it revives another monster, this enemy also has an assosciation with heavy skeletel swordsman as this is how it will know the monster it will become.



The diagram represents an object oriented system for the runes and traders within the game.

The Rune is implemented as a concrete class which inherits from item, done because it will utilise many methods from it and shares charactertistics of an item. The runes class will count how many there are within it (attribute) not that how many rune classes there are is the balance.

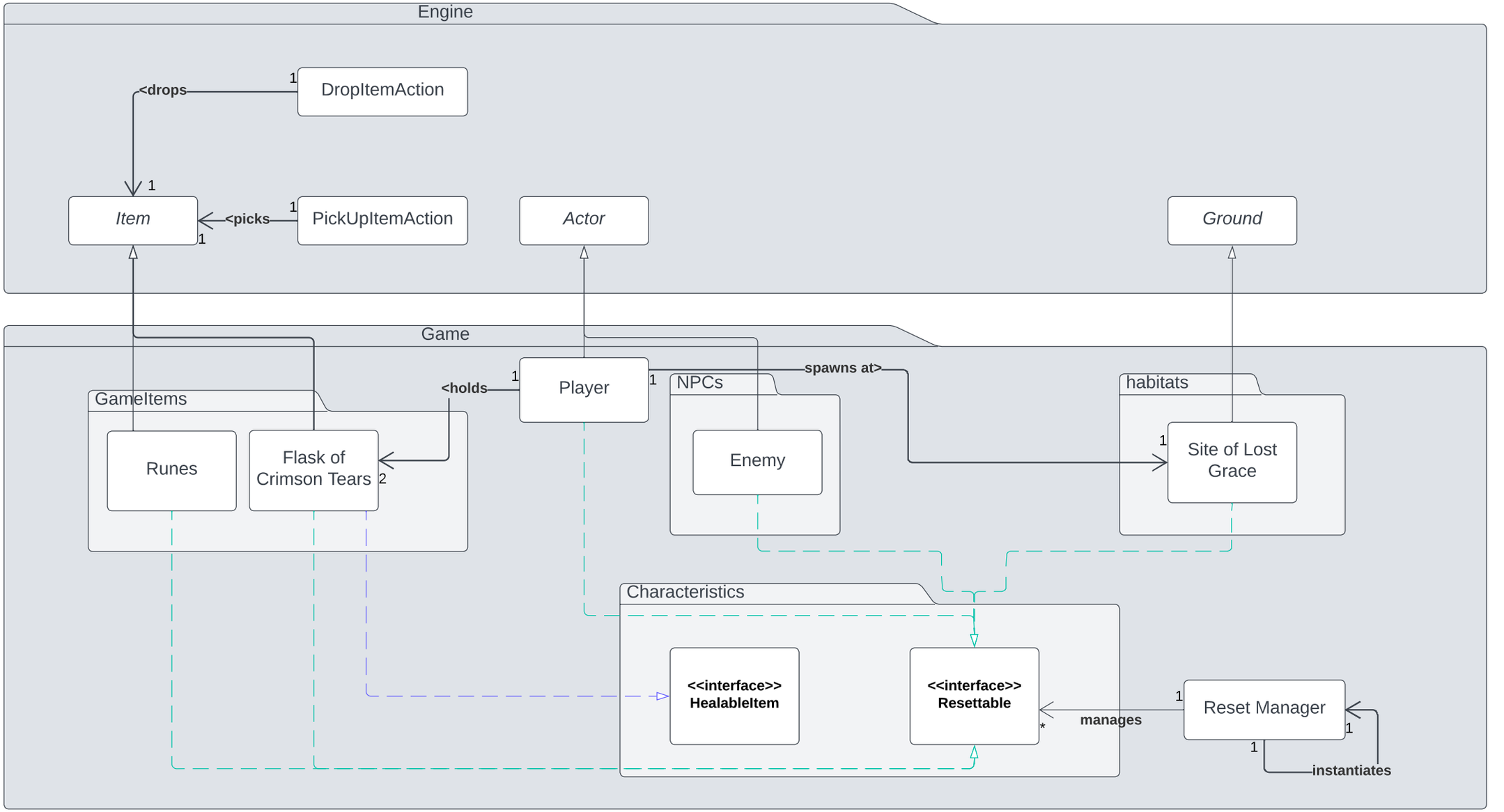
Trader will be a standalone concrete class not inheriting from actor, this is done to not violate liskov substitution principle as the trader would not fully implement the actor parent class.

The trader will have a dependency on the interface sellable, this is the dependency inversion principle as we go through the abstraction of the sellable interface rather than to each weapon which is sellable, same thing with purchasable. The interfaces are seperated to keep them small and single purporse (Interface Segregation Principle)

The trader will have an assosciation with the player as it will always sell to them and vice versa to represent the current state of 1 player and 1 trader. This could be later implemented with dependencies pointing to each other to allow for different players and different traders to be added and able to interact with each other making the code more extensible.

The enemy and player will both hold an object of runes to know how much they will drop and their current balance.

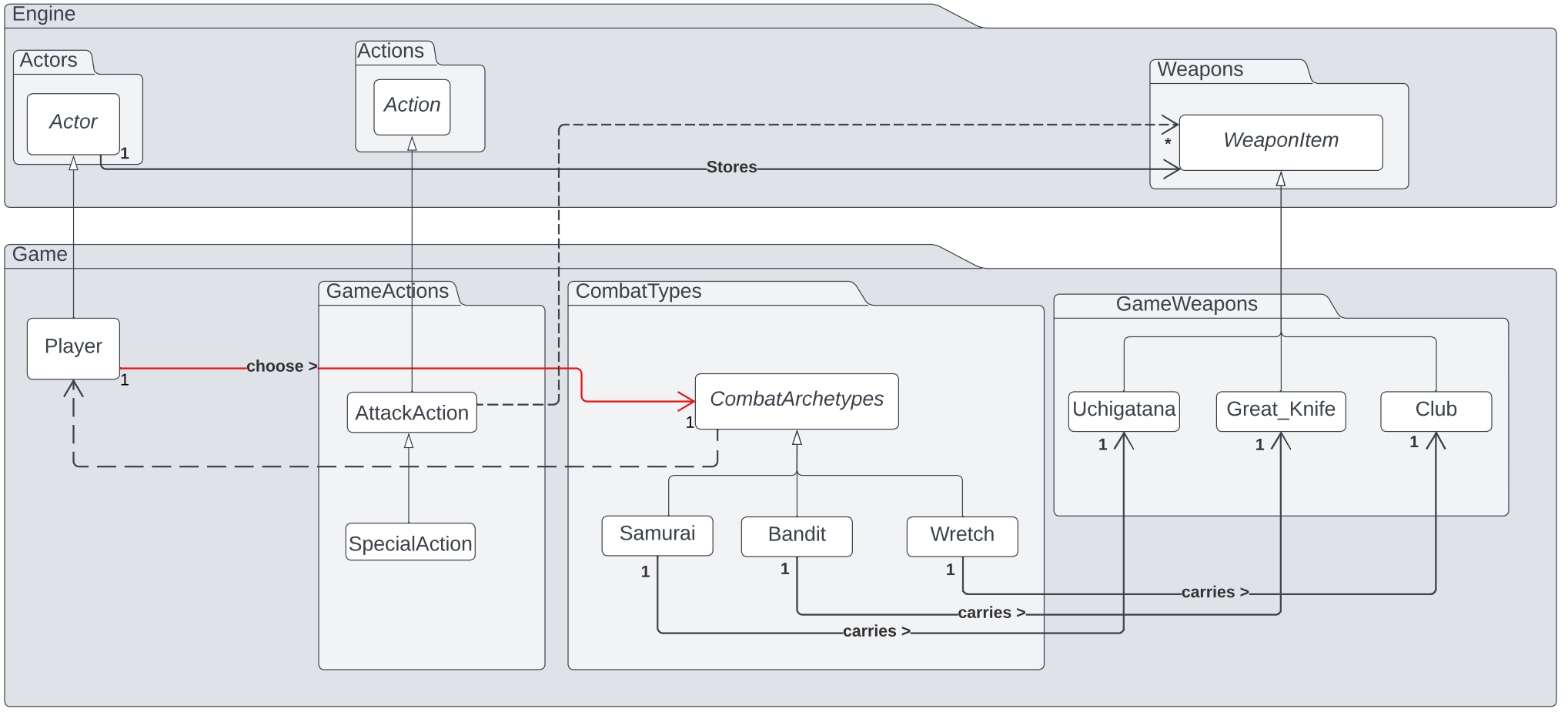
THe getRuneFromMobAction class will retrieve the enemies runes and provide them to the player this should be related with the enemy and the player, the player has an assosciation as it will go to this player always (only 1 player in system currntly), however dependency with enemy as the enemy may be different from time. This method will have a dependency with deathAction as it shall occur upon enemy death from player.



This system represents the resetting of the game and new healable items.

The Flask of Crimson tears is a subclass of Item, this is as they will utilise many of the same methods and attributes. This however can heal therefore should implement the healable item interface, this avoids multi level inheritence and as we keep the interface small and focused (Interface segregation Principle).

The reset Manager is a standalone class which only has one instance achieved by assosciating with itself. This manages all the resettable (Interface Segregation PRinciple). all resettables will inherit from this, this enables them to have unique behaviours and thus enables site of lsot grace to implement different reset functionalities. This allows the system to be more extendeable as new resettables can implement this method.



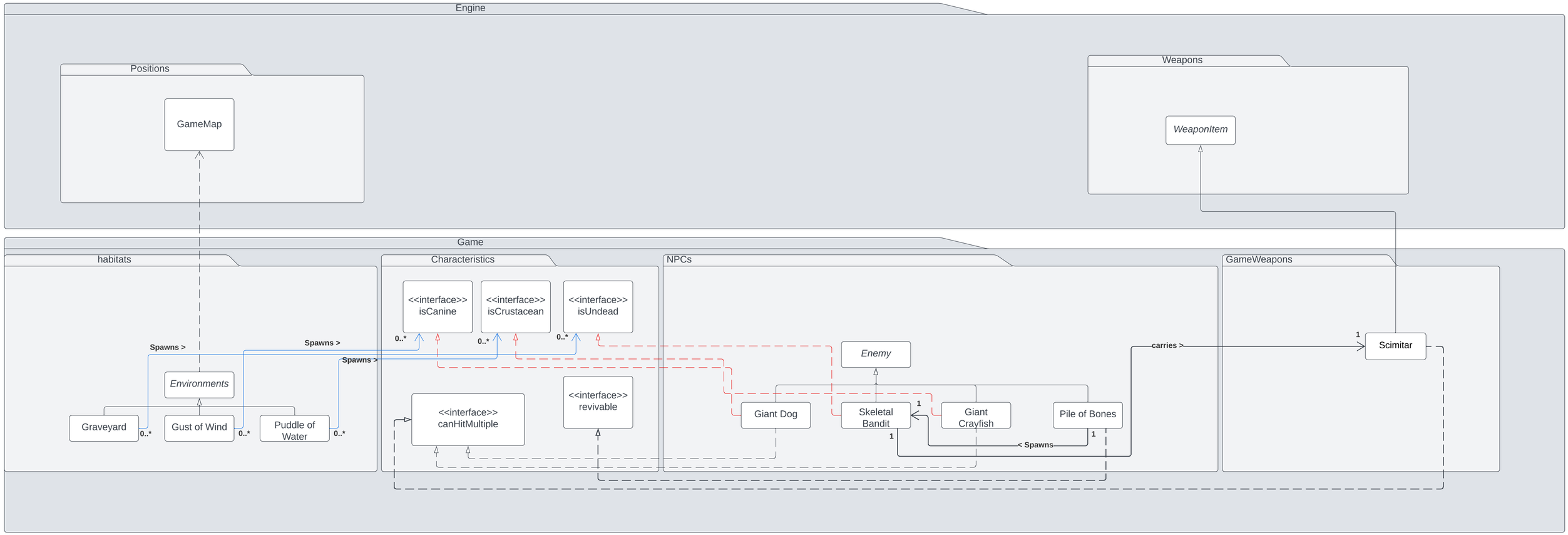
This diagram represents three Combat Archetypes which will be set up when player and three Game Weapons will be select by different player class, also this diagram represents the relations to whole system.

In REQ4, we set up new packages, CombatTypes, which include 3 Combat archetypes, Samurai, Bandit, and Wretch, and game actions. which include a pair of parents class, AttackAction as father class and SpecialAction as child class.

Due to the game requirements, the player need to choose a starting class/combat archetypes to start a game. This operation will set up player's starting hit point and weapon.

When the player choose the the Combat Archetype that they want, the game will set up the hit point and start weapon for player.

When the player try to attack the enemy, the attack action will use damage and accuracy from the weapon items to output correct damage, include the special perform.



This diagram, similar to Req 1, shows an object oriented approach for a system of enemies and environments for our game.

As the enemies that spawn from a specific environment now depends on which side of the map the environment is on, the environments now have a dependency on the GameMap class.

Three new enemy types have been added, which each implement an interface, that helps identify where the enemy spawns, and who it can attack.

As two of these new enemies have an ability to damage multiple characters in its surrounding, they have been implemented with the canHitMultiple interface.

Pile of bones has an association with Skeletal bandit, for the same reason it did with the Heavy Skeletal Swordsman sot it knows what monster to revive to.

A new weapon, the scimitar, was also added. This weapon is an attribute of the skeletal bandit, and also has the ability to hit multiple entities in its surrounding, therefore it also implements the canHitMultiple interface.