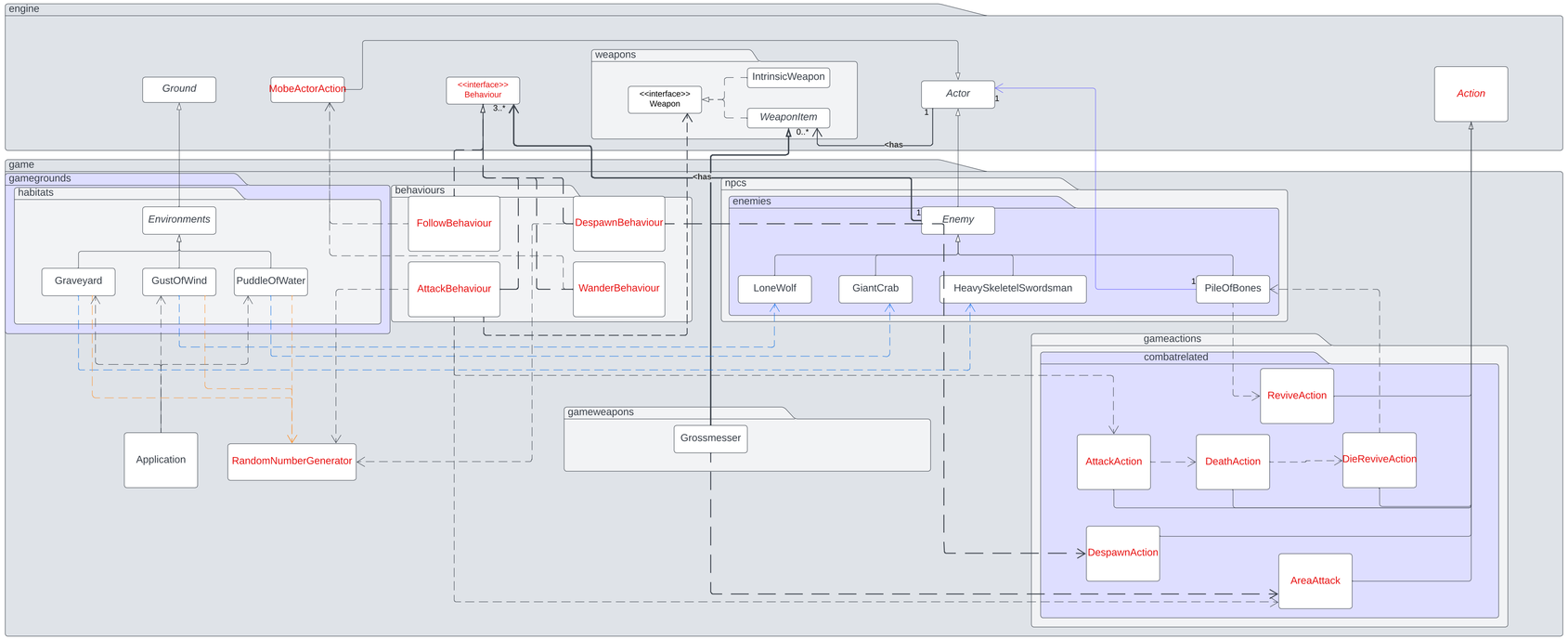
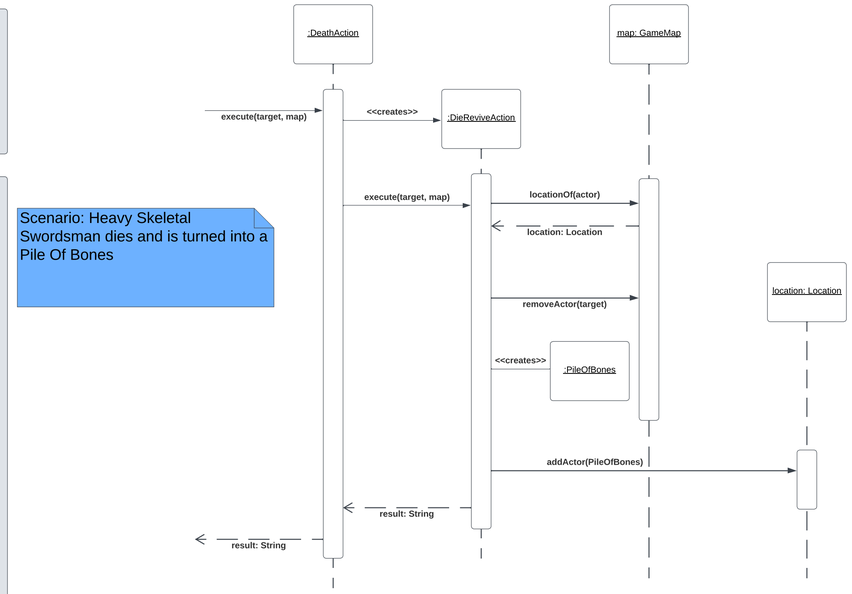




The diagram represents an object oriented approach for a system of enemies and environments in our game.

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.

Working upon our previous design from A1, we have added all the bahaviours and Actions relaevant to the enemies in this diagram.

In this implementation we have the enemy have an array of beahaviours which it will call to determine what actions it can do. This was implemented with different behaviours instead of one so that we did not violate the SRP e.g. one giant behavioutr for everything related to the enemy. This also makes it so the DIP is abided by as both high level and low level depend on abstractions.

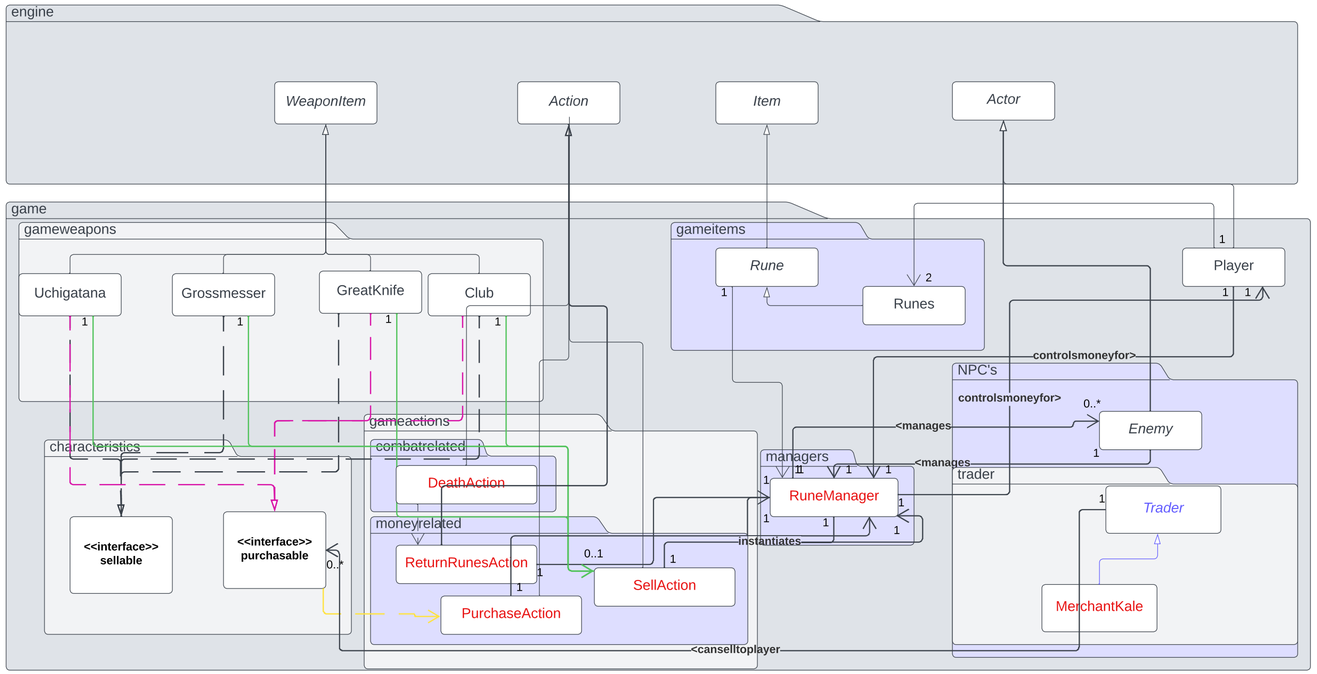
We also implment seperate Actions for many and chain them together to also abide by this SRP princeipl, this makes it easier for the code to extend as actions can be added to each other and chained to create more complex actions. This also makes it easier for the system to be open and closed as we can extend the actions by creating a new one and closed for modification as changes wont occur to old actions.

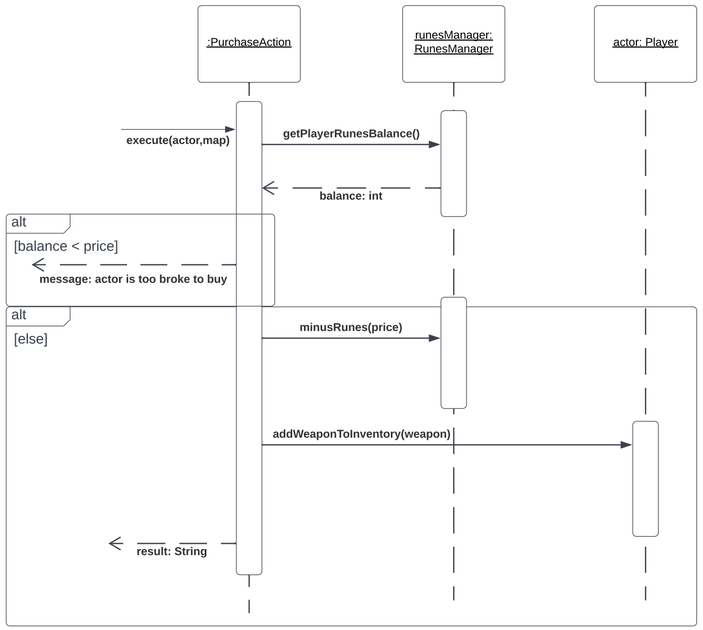
The feedback we recived let us know that instead of implementing interfaces for Area attacks and things of that nature we should instead focus on the actions so that we are able to more effectively work with the engine as well as not make the weapons too much of a god class.

For future design , the behaviours can be increased and added to the enemies, the abstraction also allows for differnt enemies to be added with different behaviours or special therefore making it more extensible.
A drawback of this design is that with the weapon providng Actions we may find it harder to differntiate just who these actions are for therefore we may have to change the methods themselves violating the Open Closed principle.

The environments could potentially be designed better as currenlty there is a lot of code repetition, this code however does not have many dependencies or use much of others and therefore are mainly fulfilling the single responsibility principle. This could be better designed through utilising a more information gathering constructor or potentially having east and west factories for the enemies

After recieving the feedback from A2 we tried to implement some fixes,
First we got rid of the magic numbers in the environment classses, as well as extracted the method out to reduce the repetition.
We removed unused attributes. We also attempted to clean up the packages and make them more informative.





Scenario: Player purchasing a weapon, and checking they have enough runes to do so

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 purpose (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.

Working upon our previous design from A1, we have added a runeManager, added actions for the trader.

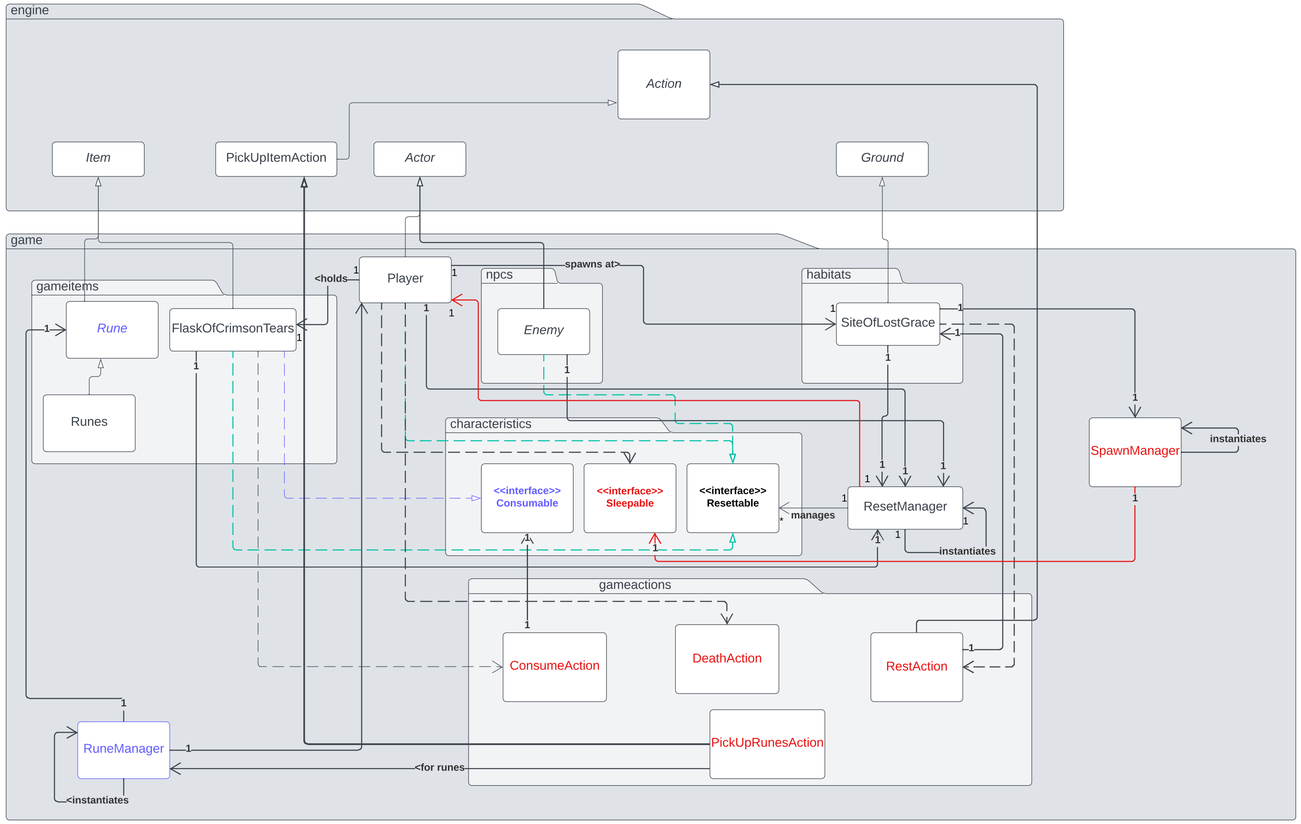
Within this req we retconned that the Trader would not beinheriting from Actor as we need to for engine classes, from feedback.
This class would be concrete, this was done as our system only had Merchantkale currently, if further traders are needed we can make a trader abstract class and have it inherit for the extensibility of the system to be furthered.

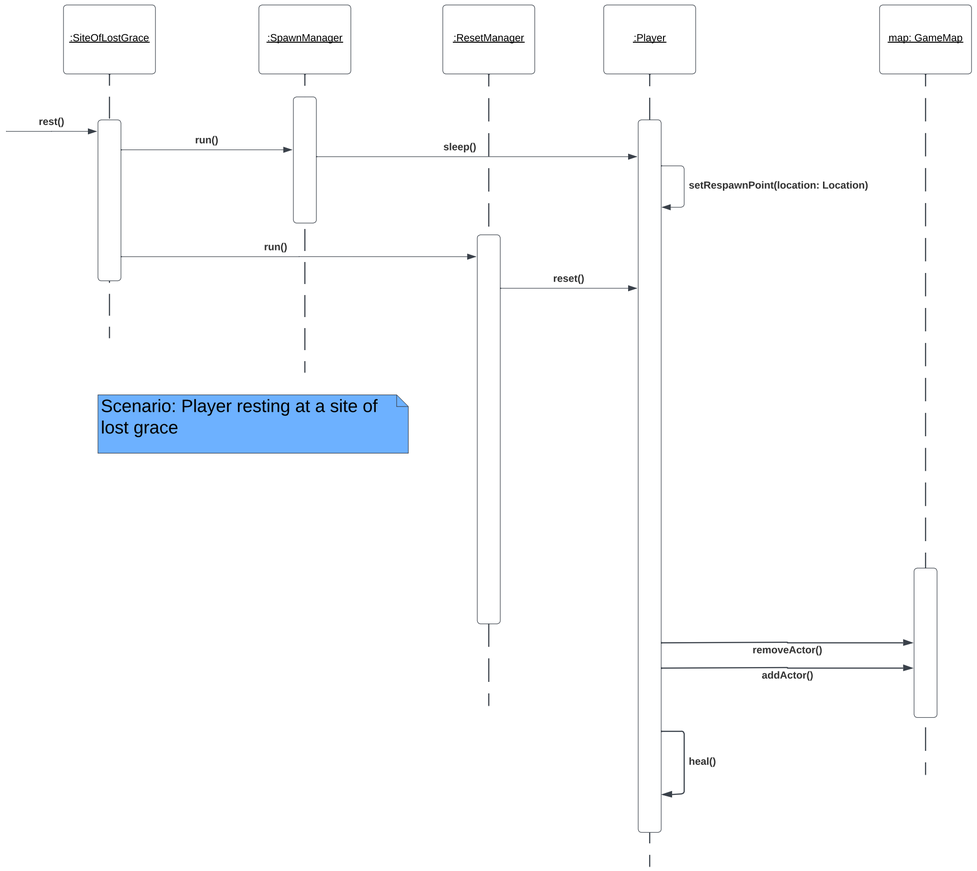
For the managing of the players runes we utilised a RuneManager which would manage the runes for a player. This is a singleton class as we only wanted one instance for the on eplayer we had. This isnt very ecxtensible as if we have more players this would need to be modified however it enables us to manage our systems players runes without downcasting in other methods and maybe leading to a bug later on.
The rune manager also controls the enemies runes, albeit a bit differntly, this enables us to implement the return RunesActions without having to have the enemy carry an item of runes, but instead only work through the RuneManager.

For the selling of items the Merchant is not involved w ith it instead going through the weapon itself, we then return it to the player as an allowable Sell Action.
This violates the DRY principle a bit as we need to repreat code for each sellable item howevr the alternative is to downcast the weapons in the trader class and make sure each item in the otherActors inventory is a sellable weapon which violates the DRY as well as also violating OCP as we need to potentially change this code often to match the different cases we may have. Therefore justifiable.

This selling and purchasing is also managed by the RuneManager as it is the only avenue we have to access the players runes, this may make it a god class if we continue adding things to it, however in this requirement its is singlersponsibility. Manage player runes.

After the feedback from assignment 2 we refactored our design a bit.
First we moved the packages around to have them more close with related ones.
The method to retrieve runes also utilises getSimpleName, which may lead to problems in the future and is a design smell however no other implmentation was thought of so we attempted to just figure a solution which allows requirements to be fulfilled and as it is a single method in the future we could refactor.
We also made the Runes inherit from an abstract rune class to enable for different monetary items to exist, therefore allows for there to be further extension. Similarly we refactored the Trader to be abstract to allow for different traders with further behaviours to be added which allowed for similar attributes to be reused and reduced repeated code.
We decided to not use find capabilities by type as according to feedback and instead use different statuses.
We also refactored the SellAction to also work with Items as well as the purchasable by using other Constructors.
 This works but there are still a few dependencies with each of the items so it could be further improved by just passsing through for example a purchasable then use a getpurchasable method which passes the item through, this however would have been to expensive to redesign given our time constraint and that we only realises this more efficient solution later.





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.

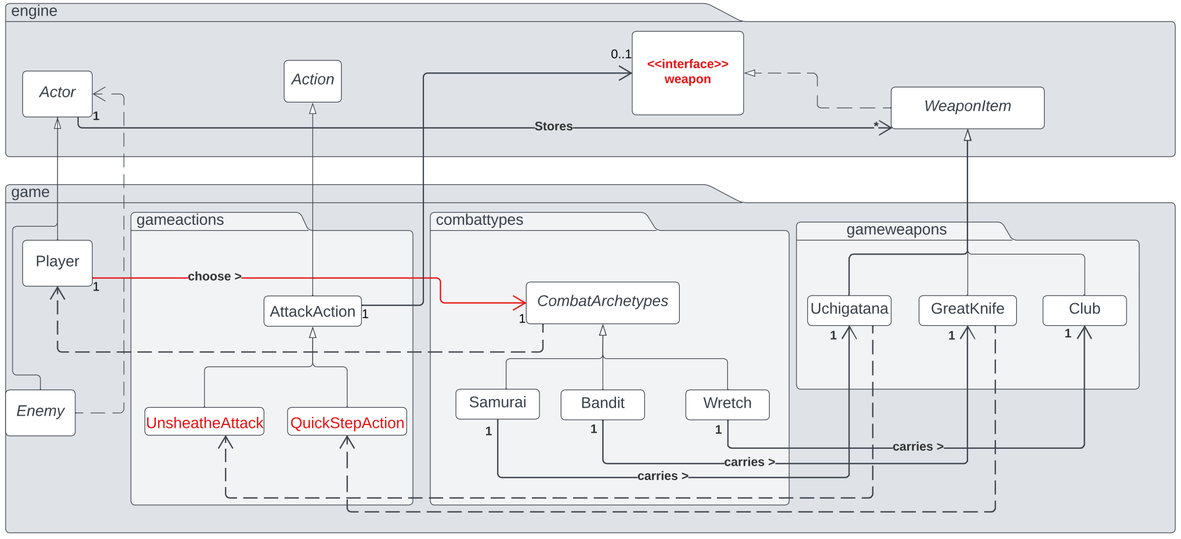
Working upon our previous design from A1, we have added the actions aswell as new singleton manager classes, Spawn and Runemanager.

We create a sleepable interface so if we ever add new things that can rest at site of lost grace we can have them abide by an abstraction and therefore abid by DIP as we have both the spawn manager and the sleepable itself depending on abstractions. Making it more extensible.

The new Actions enable all the stuff to be singleresponsibility as instead of haing the sleepable do everything we have an action to work and interact with the engine. this can help in preventing repetition DRY principle. as well as making the code more extensible e.g. more consumables we have consumeaction or resters etc.

With the introduction of new managers we enable these classes to be more specific and powerful however as we now have many manager classes all working for the player as well as the player manipulating them all it violates the singleresponsibility as player now shoulders a lot as well as the managers getting larger and larger, this can be implemented better by possibly adding more classes for it to depend on as although we normally want to RED, however it is better that than having a single class to becoming a god class which controls too much which can make it messy to work with and debug.

After the A2 feedback we changed our design.
We did not refactor the enemies to be more avstract as in our system as of yet it does not really have as much benefir as much as the cost of fixing it right now, this means we will have some technical debt in the future however the system, can work fine with the current way it is and is pretty extendable as well.
We can also utilise the resetmanager remove resettable instead of refactoring and adding each resettable as the system right now has all resettables so when we add a non resettable enemy we can remove it from resettables.
We removed the players death from the death actipon and into the players as it was having too many responsibilities when it was in the death action therefore violating SRP, this keeps our classes smaller and makes it so that we dont get a god class.
We refactored Healable to consumable as we MAY have some consumables which arent just healing items in the future.



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.

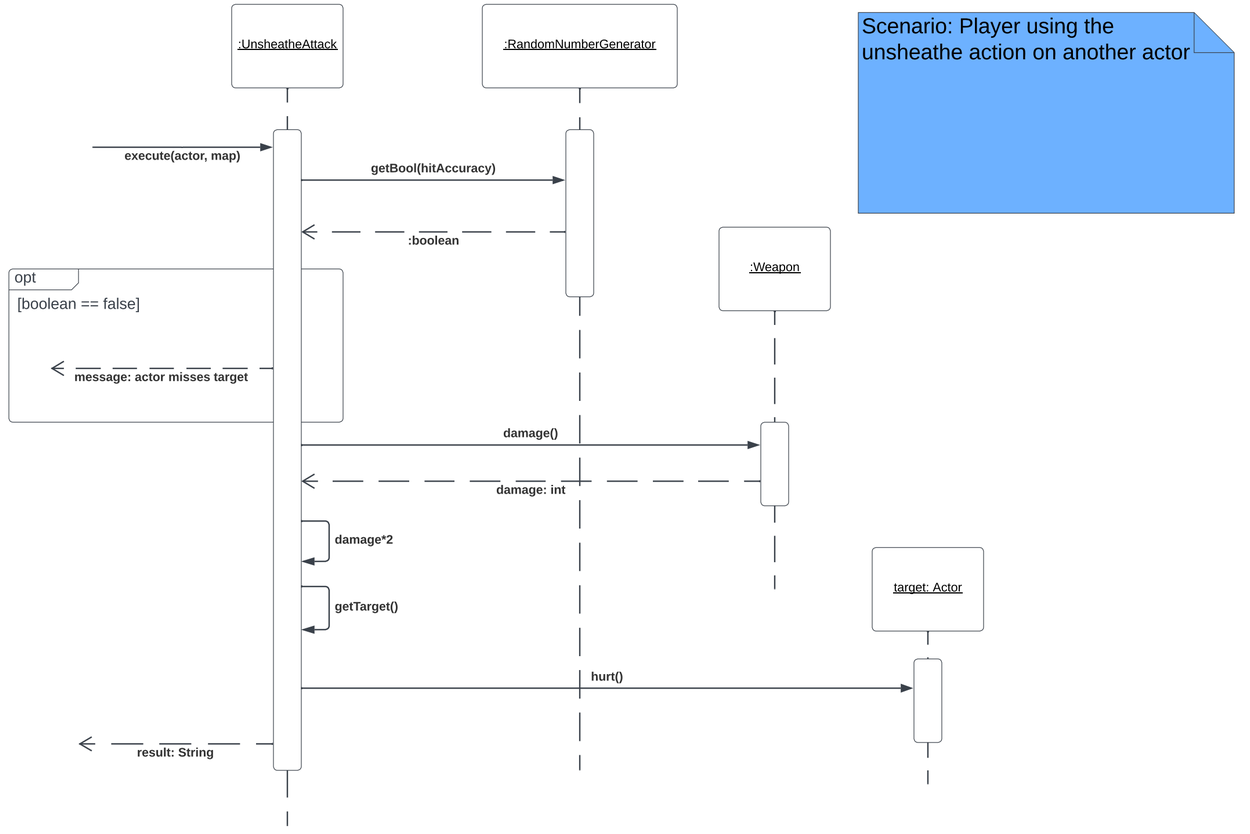
When the player choose the the Combat Archetype that they want, the game will set up the hit point and start weapon for player.

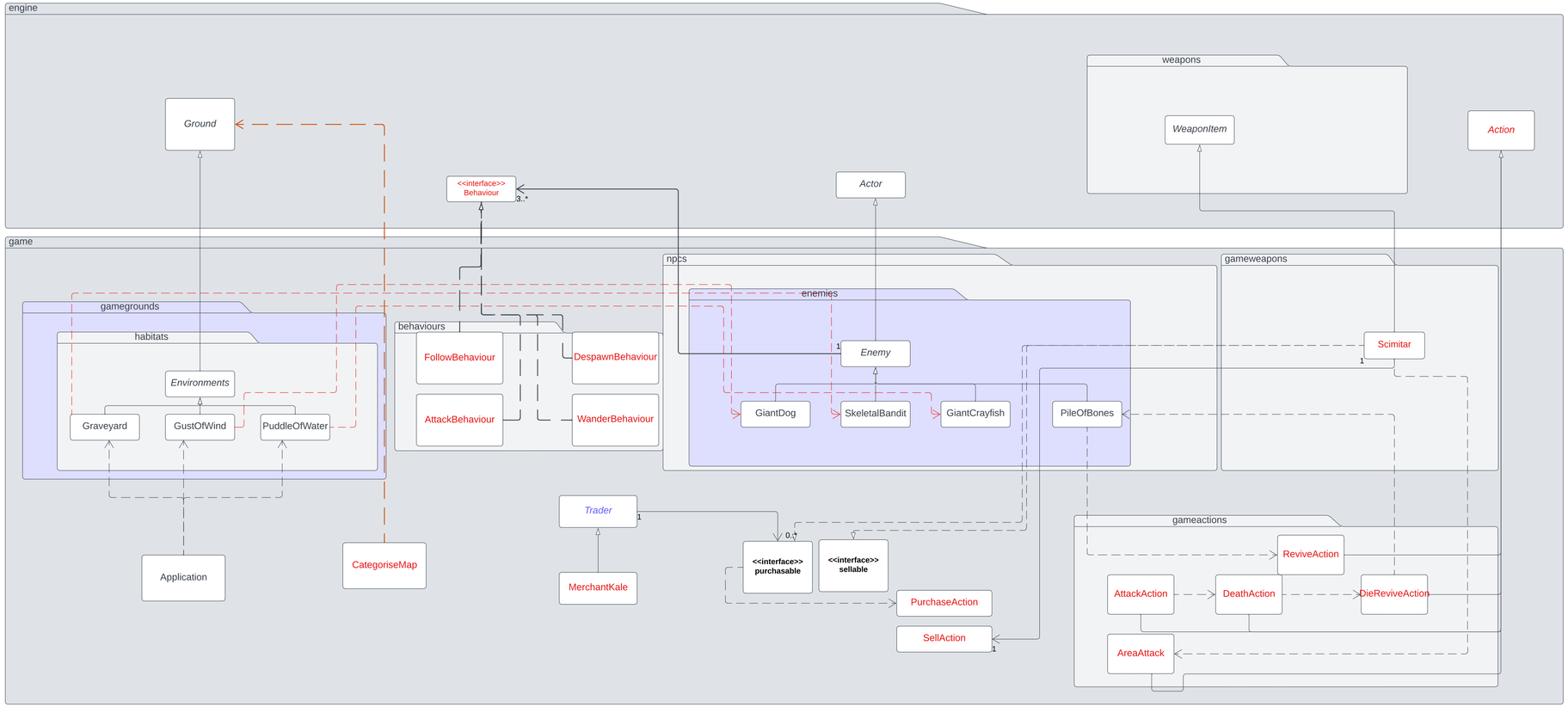
When the player try to attack the enemy, the attack action will use damage and accuracy from the weapon items to output correct damage, include the special perform.

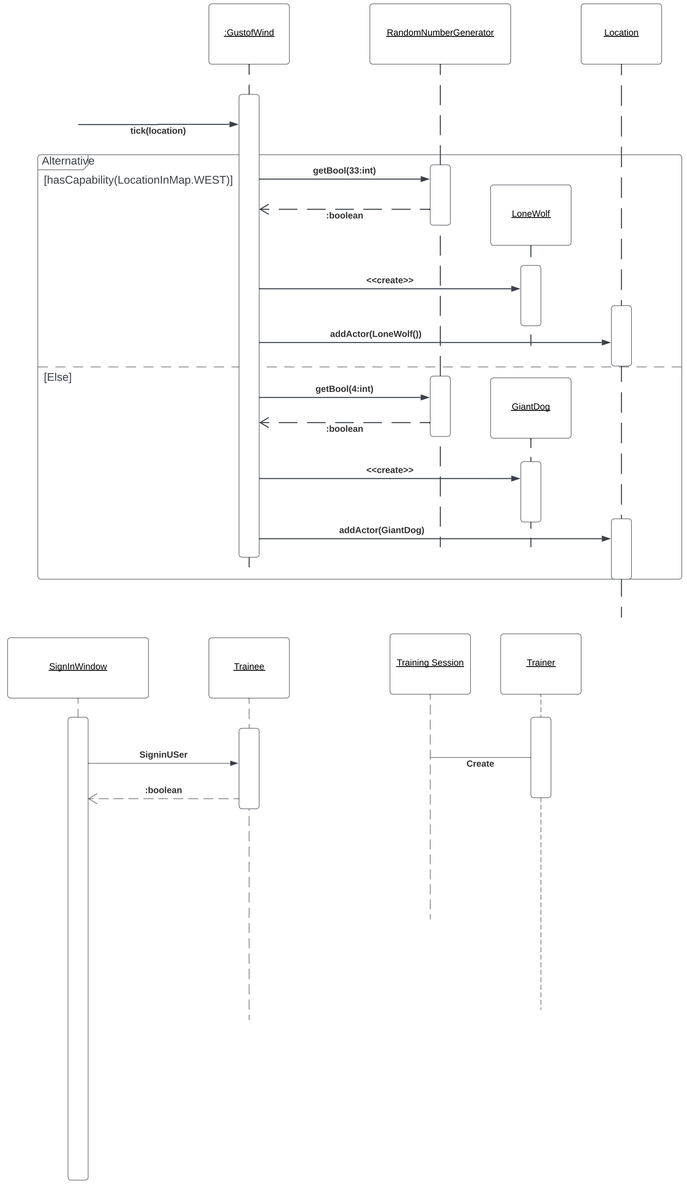
Working upon our previous design from A1, we have added the specific actions we will be employing.

These actions will inherit from the AttackAction as they all essentially attack and then provide some other benefit therefore they share some of the same attributes. however they should still be sepertate and diffrent from the others as we want to abide by SRP.

We also have the weapons from which these actions come from providing the action to the actor, this allows for there to be seperate responsibiliteis as the weapon class will provide it instead of the actor always having to determine therfore making it easier to extend as we wont have to work on the higher level stuff but instead can provide new lower level classes.







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.

Within the new design we insert the traders to include their relationship with the scimtar as well as the actions and behaviours.

This code utilises a categorise map static class to sort the map into east and west, therefore making it so we do not repeat and have to manually change them or add a check within the environment themselves which could lead to increased dependencies/assosciations.

The design builds upon the req 1 and 2 by utilising the same actions and behaviours therefore proving our code is extensible as we are able to further it by utilising the same classes. Therfefore indicating our design is polymorphic and able to work across our abstractions.

the purchasable interface is updated , the multiplicity as we now have an extra purchasbale the Scimitar.

The code is a bit repetitive as we need to add the sell action however fine as the aforementioned, req 2, tradeoff is worth it to us.