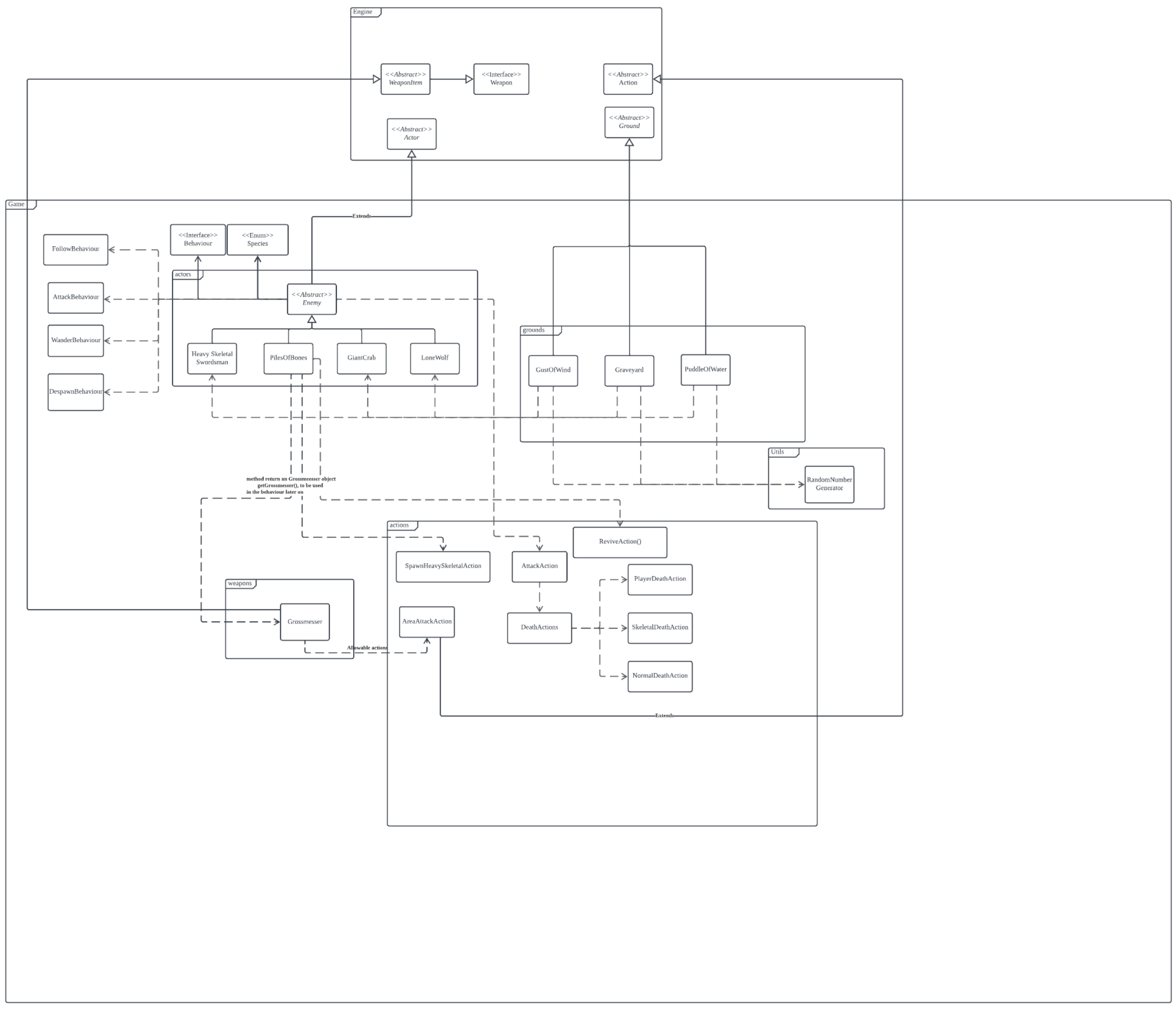
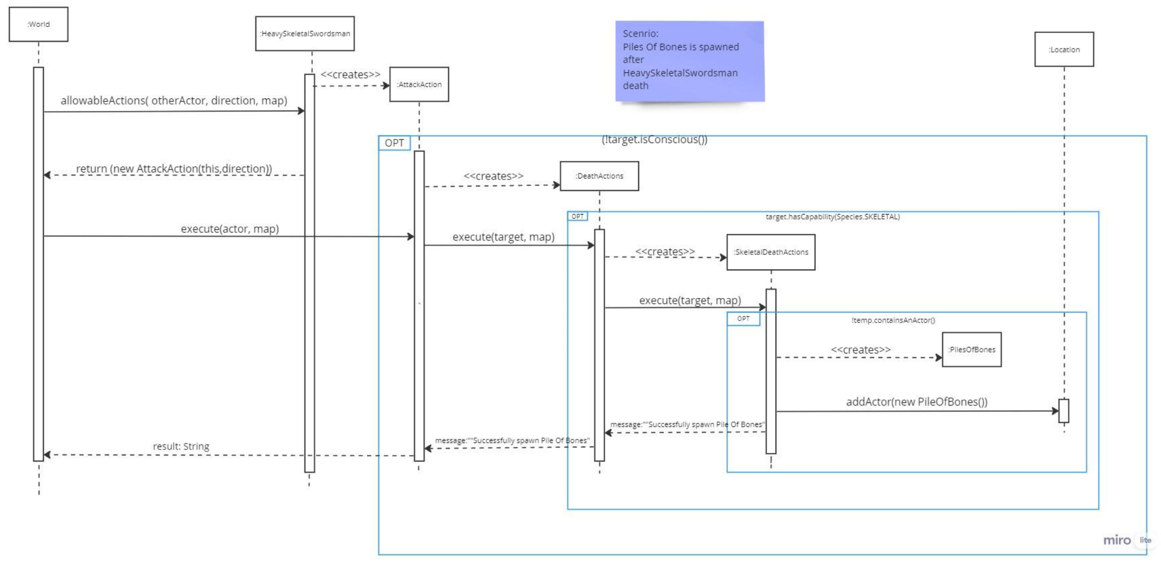
REQ1



Engine
<<Abstract>> WeaponItem
<<Interface>> Weapon
<<Abstract>> Action
<<Abstract>> Ground
<<Abstract>> Actor
Game
Extends
FollowBehaviour
AttackBehaviour
WanderBehaviour
DespawnBehaviour
<<Interface>> Behaviour
<<Enum>> Species
actors
<<Abstract>> Enemy
Heavy Skeletal Swordsman
PilesOfBones
GiantCrab
LoneWolf
grounds
GustOfWind
Graveyard
PuddleOfWater
Utils
RandomNumber Generator
method return an Grossmeesser object getGrossmesser(), to be used in the behaviour later on
actions
ReviveAction()
SpawnHeavySkeletalAction
AttackAction
PlayerDeathAction
AreaAttackAction
DeathActions
SkeletalDeathAction
NormalDeathAction
weapons
Grossmesser
Allowable actions
Extends



:World
:HeavySkeletalSwordsman
<<creates>>
:AttackAction
Scenrio:
Piles Of Bones is spawned after HeavySkeletalSwordsman death
:Location
allowableActions(otherActor, direction, map)
OPT
(!target.isConscious())
return (new AttackAction(this,direction))
<<creates>>
:DeathActions
OPT
target.hasCapability(Species.SKELETAL)
execute(actor, map)
execute(target, map)
<<creates>>
:SkeletalDeathActions
execute(target, map)
OPT
!temp.containsAnActor()
<<creates>>
:PilesOfBones
addActor(new PileOfBones())
message:""Successfully spawn Pile Of Bones""
message:""Successfully spawn Pile Of Bones""
result: String
miro lite

Assignment 2 changes:

- Sequence diagram added for scenario of HeavySkeletalDeath and spawns PilesOfBones.
- Behaviours subclasses has been specified, dependency between Enemy abstract class
- Added dependency between Grossmesser and AreaAttackAction.

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.

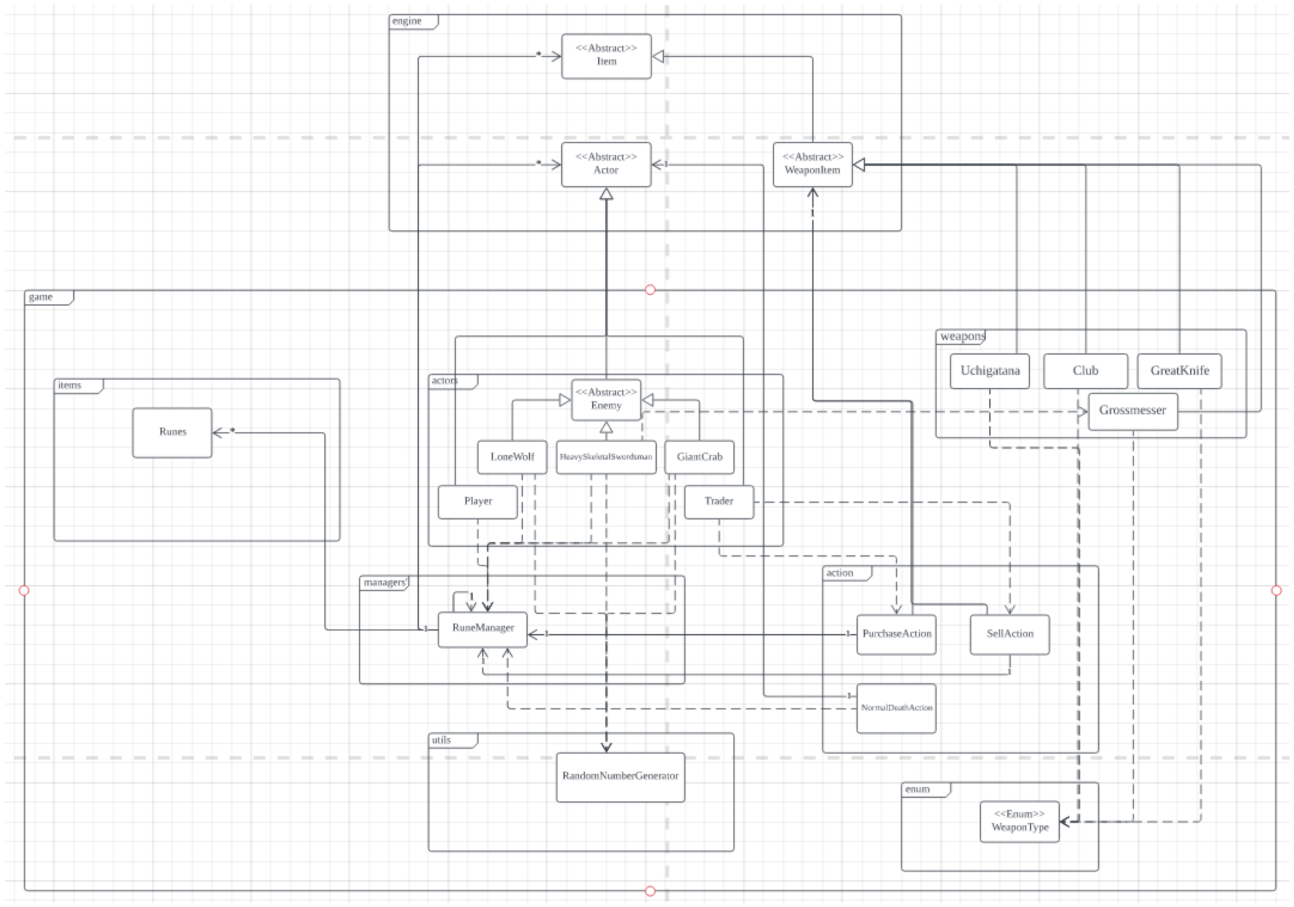
-Design Goal: Able to spawn different types of enemies from different grounds with specific probability.
All 3 types of grounds are extended from the abstract Ground class since they have common methods.For enemies , we have created an abstract Enemy class because all enemies will have similar methods.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



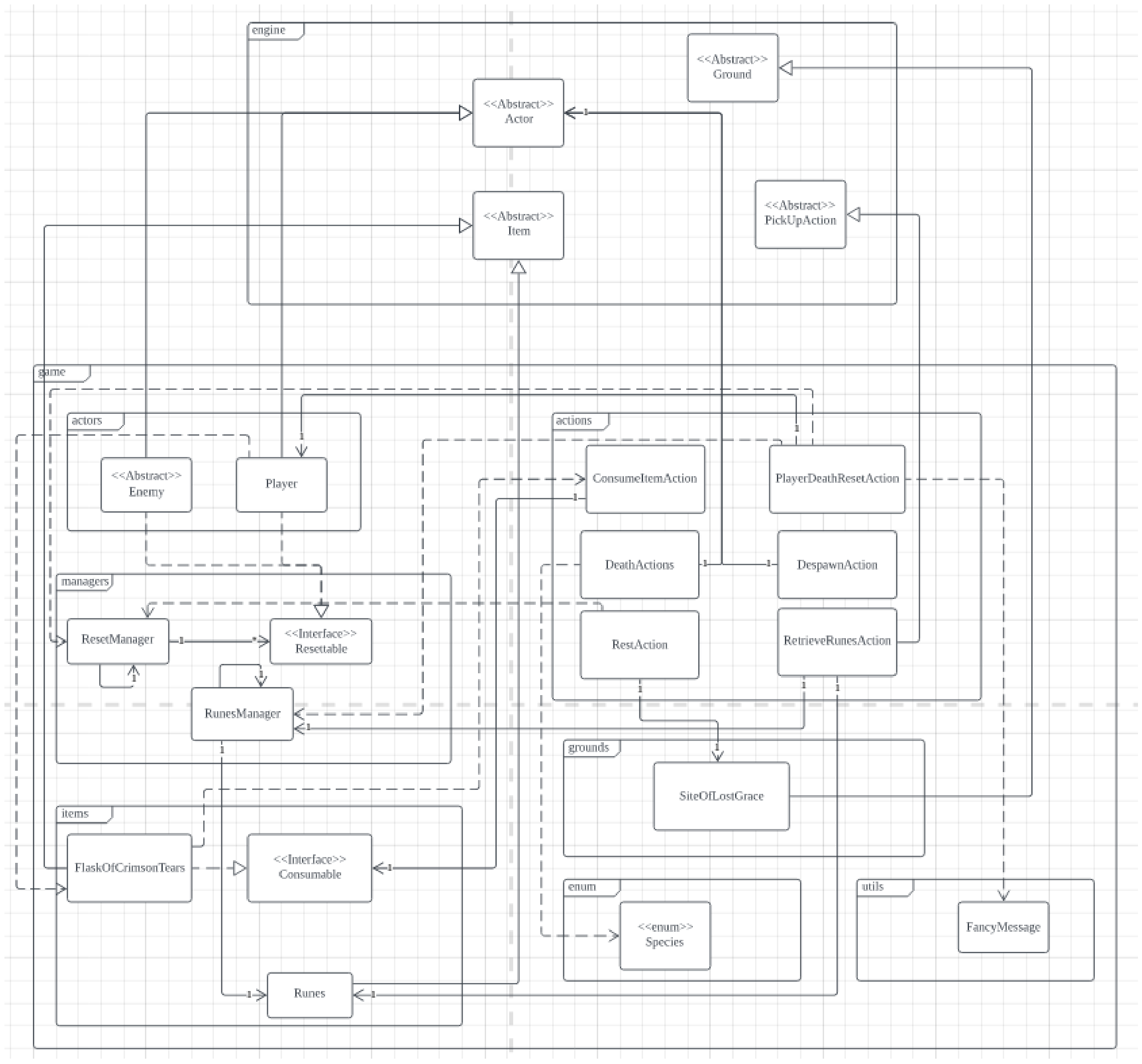
The diagram represents the interactions between traders and the players. It also includes the relationships between Runes and all Actors that are associated with the Runes class. In the game, players are able to buy/sell items when interacting with merchants/traders.
Assignment 2 main changes:
Added RuneManager class, WeaponType class.
Removed MerchantKale class.
-Design Goal:
Not all weapons are purchasable, however all weapons are sellable.
Weapons that are purchasable/sellable at Traders will be identified by adding the enum WeaponType.SELLABLE or WeaponType.PURCHASABLE into their capabilities. We avoided using Sellable/Purchasable interfaces as doing so would likely introduce the use of

instanceOf() within the Trader class to identify whether a Weapon within the Player's inventory is Sellable. Also, the usage of the enum over interfaces violates the Open/Closed principle in favour of KISS principle as all you have to do to make a weapon purchasable/sellable is add one line of code.
Purchasable weapons are inserted into the Trader's inventory after the creation of an instance of a Trader class. For now, all weapons within the Trader's inventory are purchasable however should there be an update in the future which allows Traders to enter combat by using an unpurchasable weapon, it will be easy to implement that feature with the WeaponType enum.
One downside of this design is that it assumes that all traders will be selling/purchasing the same items.

-Design Goal:
Runes are transferred upon Enemy's death, not dropped.
Instances of the rune class are not added to a Player's inventory, instead the integer attribute which represents the value of that rune instance is added to the Runes instance that is linked with the Player when an Enemy dies.
The RunesManager class is used to handle all processes concerning Runes; this follows the singleton design pattern and achieves encapsulation, however it also violates the Single Responsibility Principle as the RunesManager class will have two responsibilities: maintaining one instance of itself and also to handle all processes concerning Runes.
Since Runes are an essential component of the game, the RunesManager class will follow the eager initialization route, creating the instance of RunesManager at the time of class creation.
In NormalDeathAction, when an Enemy is killed by a player, the class will simply call a method of RunesManager, addRunes(player, runeValue) to transfer the value of the runes to the player's runes. Using a singleton in this case prevents overcomplication of the code as having multiple instances of RunesManager would introduce confusion as to which Runes instance it is referring to.

-Design Goal:
Each weapon will have different purchase/selling prices.
To implement this feature of the game, two different hashmap attributes within RunesManager are used to store the Item and their linked Runes instance. In the constructor of each individual weapon, the RunesManager methods setPurchasePrice() and setSellPrice() are called to map the weapon to their respective Runes instances which contain their purchasing/selling price; this follows the KISS principle.

REQ3



Assignment 2 main changes:
Added RunesManager class and Consumable interface.

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. The classes ConsumeItemAction, DeathActions, RestAction, PlayerDeathResetAction, and DespawnAction are all child classes of the abstract Action class within the engine, however the inheritance relationship within the UML diagram is not included to improve readability of the diagram.

-Design goal:
Player's runes are dropped upon their death, and the dropped runes will only disappear after the player dies again. Allow turn to finish and display map during player's death before executing the aforementioned steps.

To confirm that DeathActions is executed on a player, it first confirms that the actor is the player by checking for the capability Species.PLAYER, after which it will insert the capability Status.DEAD. These steps allow a separate class, PlayerDeathResetAction, to be responsible for the actions that will be executed after a player's death which achieves Single Responsibility Principle, and it also allows the game to finish the current turn.
The presence of the capability Status.DEAD is checked at the start of every turn in the Player class' playTurn() method.
In PlayerDeathResetAction, a new Runes instance is created to represent the dropped runes as in this design runes are not added to an actor's inventory, it is instead handled by a singleton RunesManager class. A private static attribute droppedRunes in RunesManager is used to point to the aforementioned Runes instance, allowing for easy accessibility and it also follows the KISS principle. Should the player die again, simply add a capability which tells the Runes class when to remove the existing droppedRunes instance from the map using its tick() method.
Should there be an update in the future that states that Runes will not despawn at all, we can simply change the tick() method and remove the droppedRunes feature.

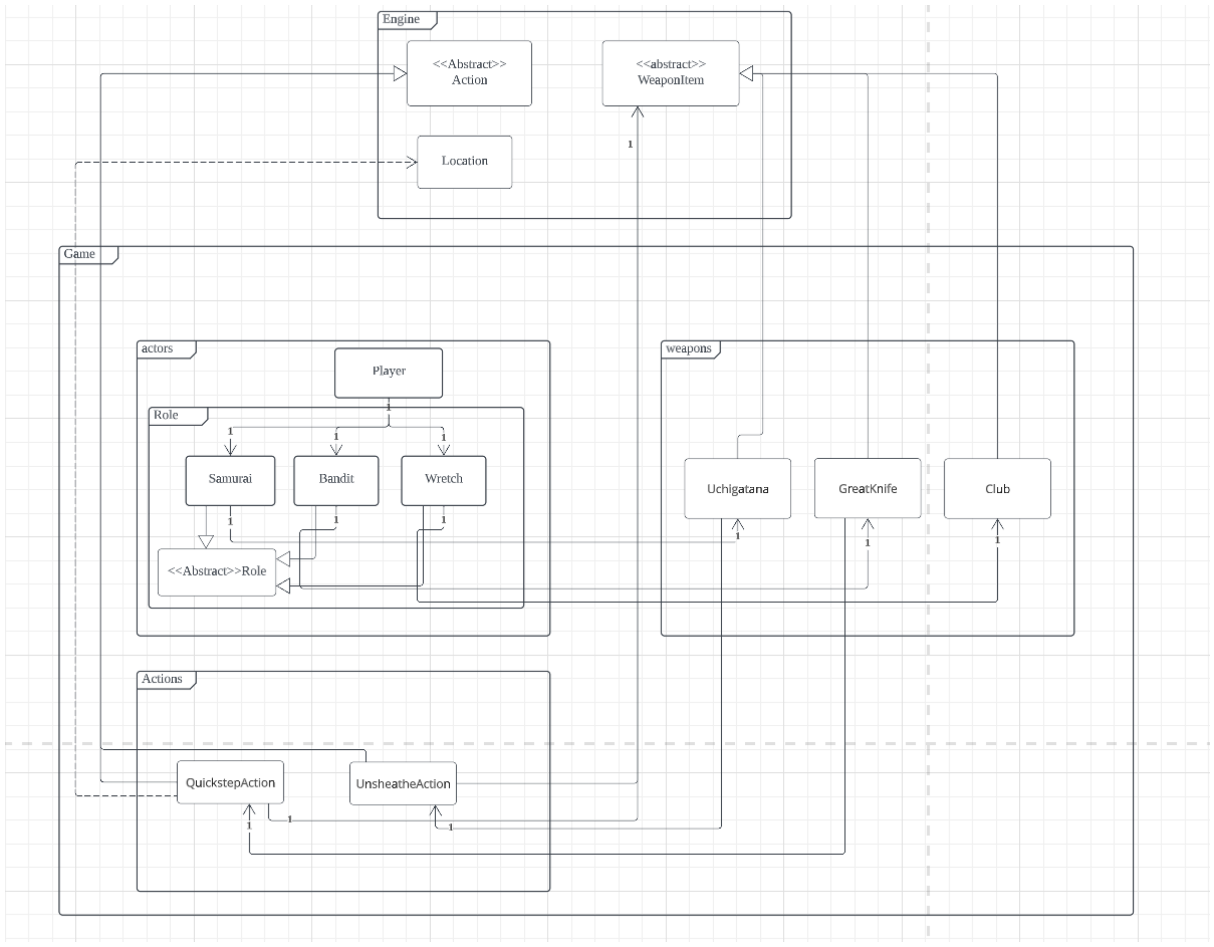
-Design goal:
After the player dies/rests, all enemies are despawned and flask usages are refilled.

To achieve DRY, the parent abstract Enemy class will implement the Resettable interface as currently all enemies will despawn after the game resets. Using the resettable interface also achieves abstraction. However, should there be an enemy which does not despawn from the map, it will be easy to implement that feature just by overriding the parent's reset() method within the child class.
A separate class DespawnAction is used to remove the actor from the map, this follows SRP as it only has one responsibility.
The FlaskOfCrimsonTears class will also implement the Resettable interface.
To manage all resettables, a singleton ResetManager instance is used, although this violates SRP, it enhances the simplicity of the code as we can just call a method of ResetManager to execute the reset() methods within all resettable instances.

-Design goal:
Consumable items will each perform separate actions when consumed.

Instead of having consumable items inherit from a parent ConsumableItem class, they will instead implement the Consumable interface. This improves the level of abstraction within the design, and also improves code extensibility.
In ConsumeItemAction's execute(), the implemented methods of the consumable interface will be called, in a way this achieves OOP such that we don't need to edit the ConsumeItemAction's execute() method to change the functionality of a consumable.
It will be very easy to implement additional functionality to an individual item's consumeItem() in the future.

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.

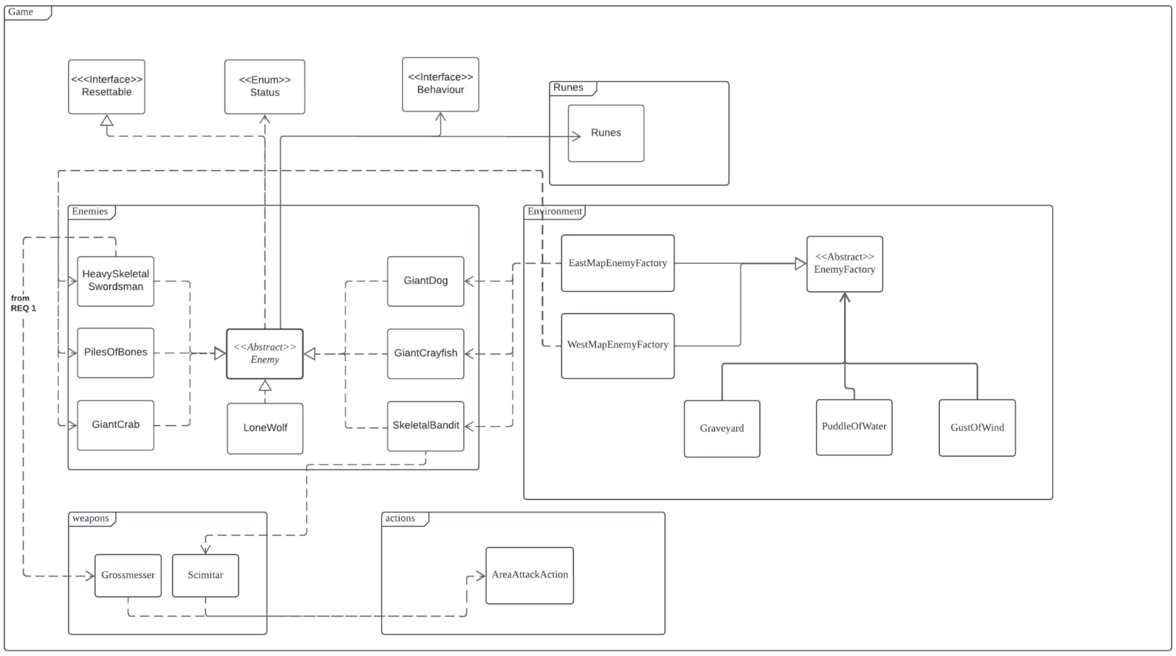
All 3 weapons extended the abstract WeaponItem class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY).

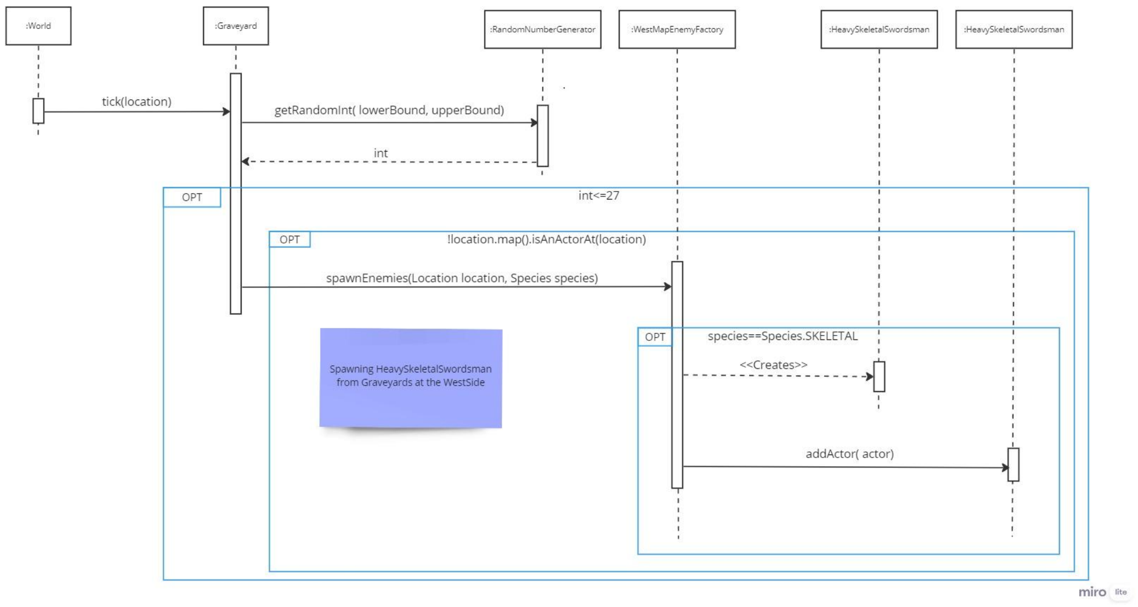
All 3 weapons classes will have a dependency on the respective starting class because those are the starting weapons of those classes so making them attributes of those classes is necessary .

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




:World
:Graveyard
:RandomNumberGenerator
:WestMapEnemyFactory
:HeavySkeletalSwordsman
:HeavySkeletalSwordsman
tick(location)
getRandomInt(lowerBound, upperBound)
int
OPT
int<=27
OPT
!location.map().isAnActorAt(location)
spawnEnemies(Location location, Species species)
Spawning HeavySkeletalSwordsman from Graveyards at the WestSide
OPT
species==Species.SKELETAL
<<Creates>>
addActor(actor)
miro lite

Assignment 2 changes:

- Sequence Diagram added
- New abstract EnemyClass introduced, WestMapEnemyFactory and EastMapEnemyFactory is the extension of it. They are responsible for spawning different enemy type based on their orientation(West and East)
- CheckOrientation() class is removed

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)