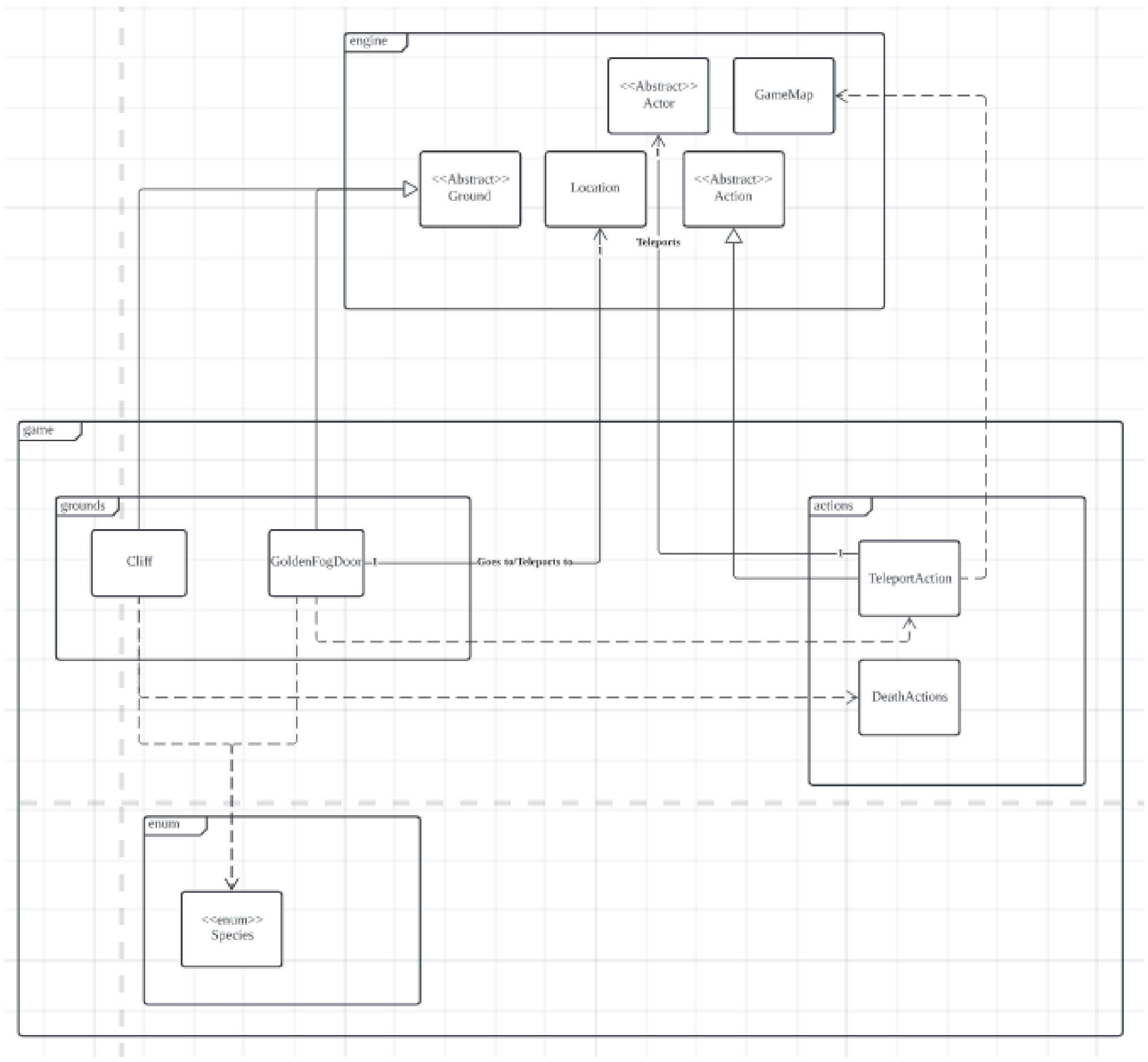
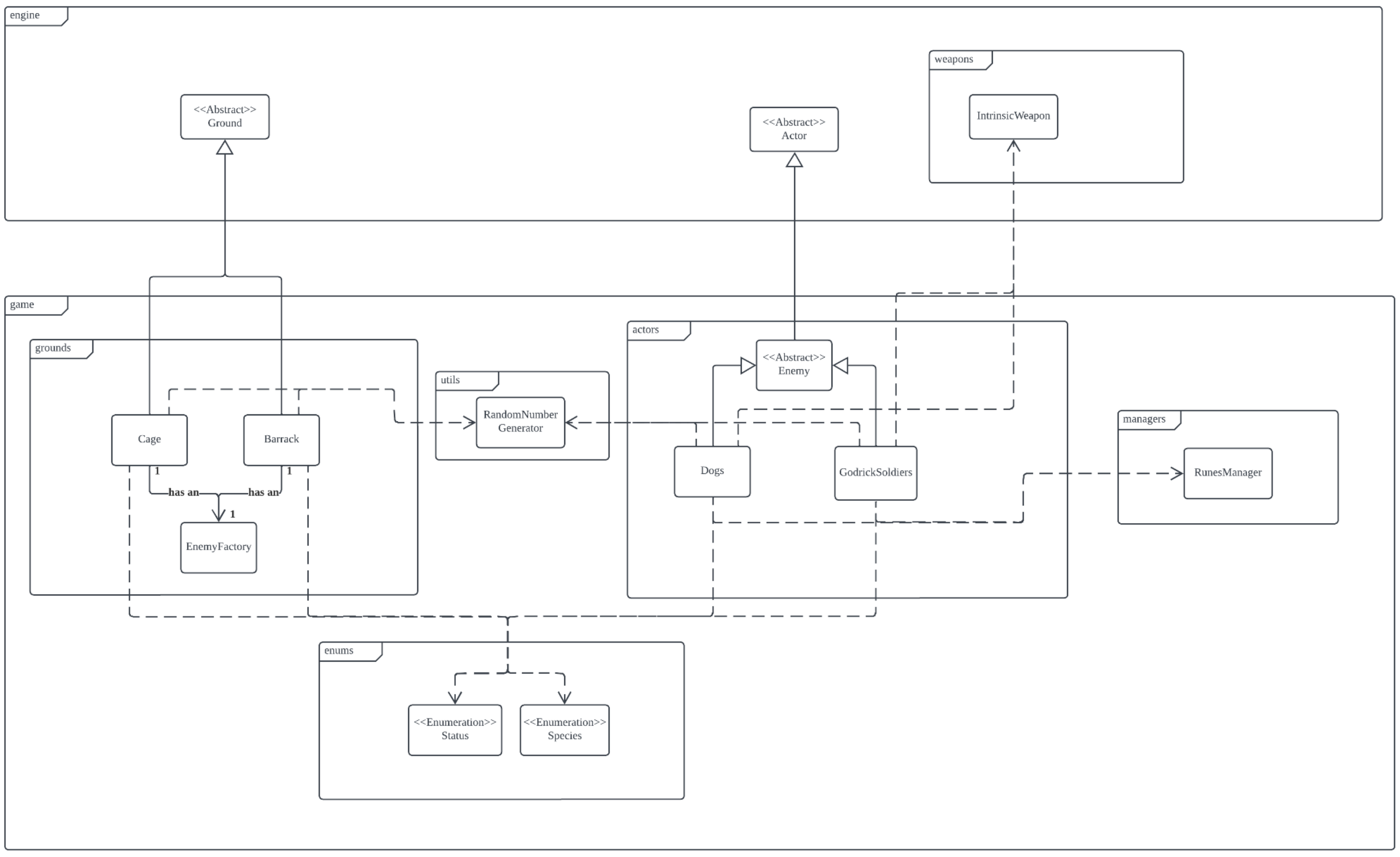
REQ1



The diagram represents the interactions that are given by the following grounds: Cliff and GoldenFogDoor.
The main changes that were done after receiving feedback from Assignment 2 is:
Include name and multiplicities for association relationships.
-Design goal:
Only the Player will die when they step on a Cliff, other Actors will not.
To accomplish this feature in our design, only players are allowed to step on the Cliff. This is done by using Ground's canActorEnter method and the Species.PLAYER capability to check if the Actor is a player.
The Cliff's tick method will have an if statement checking if there is an actor in their location, since only Players are allowed to step on the Cliff, DeathActions will instantly be called on the Actor that is currently on the Cliff.
This implementation follows the KISS principle as there is no complicated logic/relationships taking place to accomplish the design goal. It also follows SRP (Single Responsibility

Principle) as the Cliff class will only have a single reason to change: The logic concerning which actors are allowed to step on and fall off the cliff (in canActorEnter).
The Cliff class also follows the LSP (Liskov Substitution Principle) as its constructor takes in no parameters, thus it is able to be used by existing classes within the engine such as the GroundFactory.
-Design goal:
Only the Player is allowed to use Golden Fog Doors to traverse between maps.
To accomplish this feature, there will be logic done within the canActorEnter method that checks if an actor has the Species.PLAYER capability before allowing/declining the actor from stepping on the ground.
A separate class, TeleportAction is then used to handle the action of a player teleporting to another map. This follows the SRP (Single Responsibility Principle) as TeleportAction is only concerned with the teleportation of a player.
The GoldenFogDoor class will also store the destination location of the opposite Golden Fog Door when it is initialised in the Application class, this is passed into the constructor of TeleportAction. This follows the KISS principle as you can easily change the destination of the GoldenFogDoor when initialising without having to modify any existing code.
-Pros:
Since the Cliff class follows LSP, it'll be able to be used by the FancyGroundFactory class, thus cliffs can be easily added onto future maps without the need for the usage of setGround.
Since the action of teleporting is not handled by GoldenFogDoor, the TeleportAction class can be used in the future by different classes to let different actors teleport.
-Cons:
GoldenFogDoor does not follow LSP, therefore it will be difficult/time-consuming to add golden fog doors onto different maps as you would have to find the coordinates first before being able to add them.

REQ2



The UML diagram represents the newly implemented ground and enemies.

-Design Goal: Create new ground that is responsible for spawning new species of enemies.

We created this new ground just by extending from the abstract Ground class in the engine package,
and provide it with a new name and new display character. For these new grounds it will spawn the same
variant of the enemy regardless if the ground is located at east or west. If in the future new variant
of the enemy is given, we can just add it to the EastMapFactory or the WestMapFactory.There is literally
zero modification to the existing code in order to implement this feature. We have chosen this approach

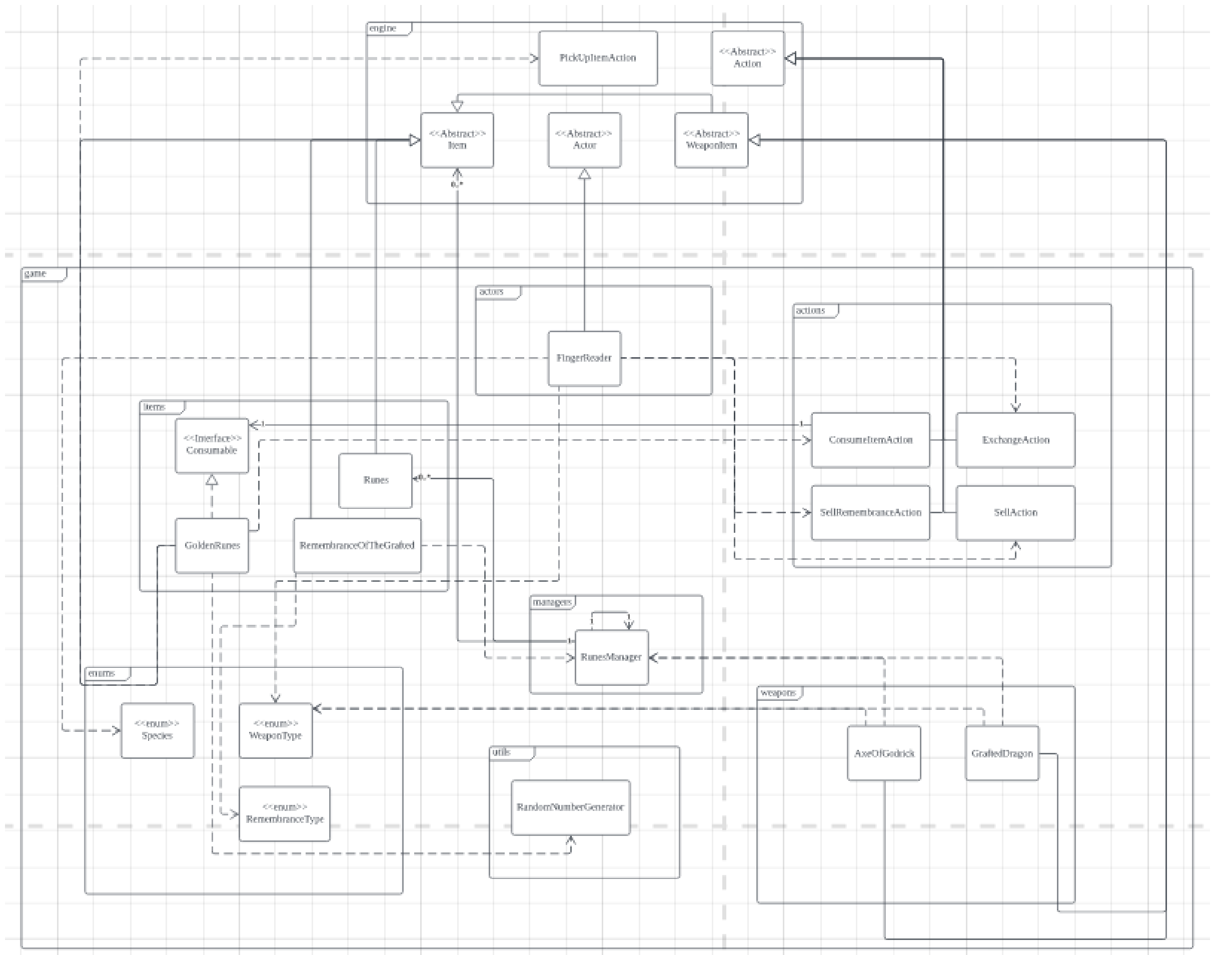
because it follows the OCP-Principle(Open-Closed Principle), as shown above we can add as many grounds as we want. (Open for extension)

Other than that, this approach also achieve the SRP Principle(Single Responsibility), since each of the ground
class are only responsible to call the EnemyFactory's spawnEnemies() method, and the handle will handle the enemies
spawning regarding the location of the respective ground in the map.

-Design Goal: Create new enemies, which will spawn respective to the ground.
We created these new enemies just by extending from the abstract enemy class in the game's actor package.
As all the enemy behaviour is the same, we have coded it in the abstract enemy class, so that we do not have to rewrite the same behaviour for every new enemy created. We just need to give new enemy a new name,
new species, new display character and Intrinsic weapon. We will add weapon to weaponInventory if any new
weapons are provided. This achieve the DRY(Don't repeat yourself) and OCP(Open-closed principle) at the same time
as we only code the enemy behaviour once, any new behaviour for specific species, we will just add it to the specific
species without modifying the existing code.

-Pros:
Since we follow OCP Principle we can create even more enemies in the future, just as easy as extending from the Enemy abstract class.
-Cons:
When a new variant of the same species is added, even though OCP Principle is applied, we still have to add that variant into the EastMapFactory and WestMapFactory according to the requirement.

REQ3



The diagram represents the interactions involving the classes FingerReader, GoldenRunes, RemembranceOfTheGrafted and the new weapons AxeOfGodrick and GraftedDragon. Changes that were applied to the UML diagram after the feedback from Assignment 2 is mainly adding names and multiplicities to association relationships.

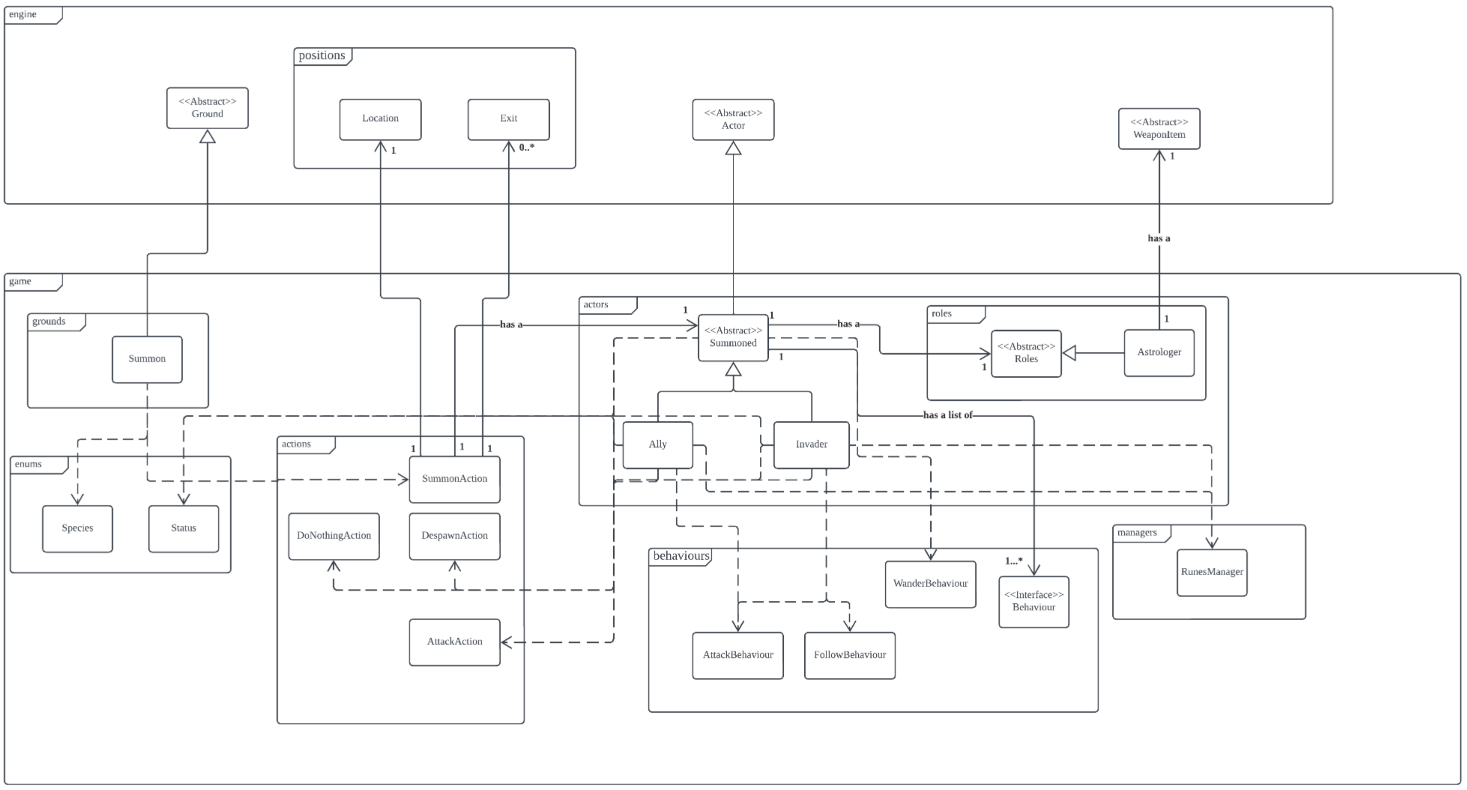
Design goal:
Golden runes can be consumed after being picked up, it can also be dropped.
The GoldenRunes class will implement the Consumable interface which allows ConsumeItemAction to be performed using the GoldenRunes. This follows the DIP(Dependency Inversion Principle) and also the OCP(Open-Closed Principle) as we wouldn't need to modify the existing ConsumeItemAction class to allow the Player to consume GoldenRunes.

Design goal:
Remembrance of the Grafted can be exchanged for Axe Of Godrick or Grafted Dragon at Finger Reader Enia, or sold for 20000 at any trader/reader.
The RemembranceOfTheGrafted class will have the RemembranceType.GRAFTED capability, it will have a dependency relationship with RunesManager in order to set its selling price to 20000. The use of the capability avoids the usage of instanceOf as all

Traders and FingerReaders will check for this capability before allowing the Player to exchange or sell said item. OCP is violated in favour of KISS because of the usage of capabilities as we had to change the existing code within the existing Trader class to implement this feature.
If there were to be more remembrance types in the future that can be exchanged for different types of weapons, we can create new capabilities in RemembranceType to allow future remembrances to be exchanged for different items.
Separate, and new, classes ExchangeAction and SellRemembranceAction are used to contain the code for exchanging the Remembrance for new weapons or to sell the remembrance for 20000. These Action classes follow SRP(Single Responsibility Principle) and LSP(Liskov Substitution Principle) as these classes only have code related to their name and they are also able to be substituted for their parent Action class.

-Pros:
The RemembranceType enum class allows future Remembrance items that are not Remembrance of the Grafted to be identified. This allows them to be exchanged for different items.
-Cons:
Future ExchangeActions involving different weapons will have to change existing classes due to our decision to use capabilities to identify Sellable/Purchasable/RemembranceType objects.

REQ4



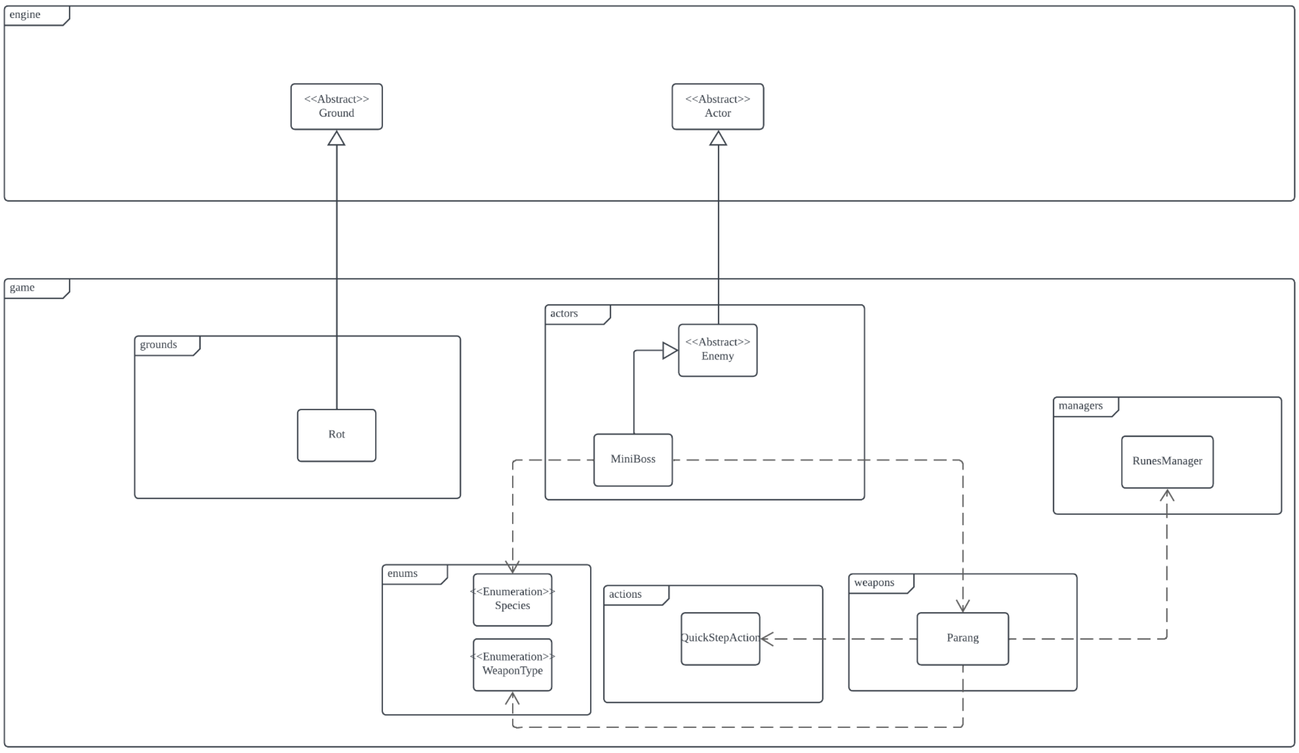
The UML diagram represents the new type of actor created, which is the summoned actor. They will spawn from the summon ground
The game has introduced a new feature where the player is allowed to spawn a new Ally/Invader at a fixed possibility at summon ground.

Design Goal: Implement new actor, Ally/Invader
As Ally and Invader have very common behaviour, we have created a new abstract class just for actors being summoned, Summoned abstract class.
The other approach is to have one Ally class and one Invader class and without an abstraction class which will violates the DRY principles, because both actors only have a minor difference in their behaviour.

Design Goal: Implement a new ground, that is responsible for spawning Summoned actor
As it is just another ground, so we created a new ground class called "Summon" which is extended from the abstract ground class, to achieve the DRY principle.
In order to keep to the SRP principle , the ground is only responsible for returning a SummonAction(), instead of directly handling the spawn of a Summoned actor.
Inside the SummonAction() we have applied the Dependency Inversion Principle(DIP). This is because when we spawn the actor, we depend on the Summoned abstract class, instead of having direct dependency to the Ally and Invader class. With this principle, we do not need to have separated spawn method for Ally and Invader which will violate the OCP principle in the future if a new type of Summoned actor can be spawned from the ground.

-Pros:
As mentioned above the SRP Principle is applied in the ground class, so the ground “Summon” doesn’t directly handle the spawn of an actor but is handled by the SummonAction() instead. In the future, if we have any feature that is required to spawn an Summoned actor, we can reuse the SummonAction()
-Cons:
Since SummonAction() are only capable of summoning Ally/Invader which has something in common, if any feature requires them to summon something different they may need to create a new summonAction responsible for summoning another type of actor.

REQ5



The UML diagram represents a new map created that is consist of new type of ground with special effects if actor standing on it,
also a MiniBoss is introduced in the map.

Design Goal: Player able to defeat the MiniBoss and get a new weapon that is drop by the Miniboss, at the same time the player needs to avoid the ground "Rot" that will damage the player if standing on it.

Even Though, a new type of ground which will have a status effect on the actor standing on it, we can still extend from the abstract ground class, which demonstrates the OCP Principle. The MiniBoss which behaves a little different from the rest of the enemy can still be extended from the abstract enemy class, the existence of this MiniBoss does not need to modify the existing code even though it behaves differently.

This new Mini Boss has a new type of weapon in his inventory , "Parang" which is also extended from the existing weaponItem interface.
This special requirement is to demonstrate that the OCP principle does not only apply to any specific class in the game, it's applied to every object in the game.
This whole requirement does not modify any of the existing code.

-Pros:
Since we follow OCP Principle we can create even more enemies in the future, just as easy as extending from the Enemy abstract class.
-Cons:
But when it comes to an enemy that is not spawn from the ground, we need to get the exact coordinate of the location that the enemy needs to be, and manually add it to the map inside the Application