# REQ 1 - DESIGN RATIONALE

New classes in our implementation:

- Cliff
    - FallAction
- Teleporter
    - Golden Fog Door
- MapManager

In this implementation, our new ‘Cliff’ class extends the ground class provided by the game engine, and is modified to return a new ‘FallAction’ if an actor (who is not null) stands on it. This is in line with the single responsibility principle as the cliff is solely responsible for its displayChar and whether or not an actor is standing on it; and if it does, then returning the fall action. Continually, as Cliff extends ground, this follows the DRY principle.

The creation of the new FallAction class therefore follows the interface segregation principle and the single responsibility principle, as it is solely responsible for when an actor falls to their death. It is also open for extension as the FallAction class can be implemented on other environments that may involve an actor falling to their death, not just the cliff class. This follows the OCP principle, as this action can be extended to other environments; but is closed to modification.

The Teleporter class extends the preexisting Ground class, provided by the game engine. This teleporter acts as the ‘door’ between maps; which the GoldenFogDoor class extends. This teleporter class follows SRP as it is solely responsible for checking if the actor (presumably the player) is able to move between maps; and thus teleporting them from map to map. It also follows the Liskov substitution principle, as parent classes of the player are able to move between maps; specifically the Actor parent class. This also follows the dependency inversion principle as the Teleporter class is abstract; meaning that many child classes can extend it and maintain its teleport method while having a different displayChar. This is exemplified by the GoldenFogDoor class, which exists only to define its displayChar as ‘D’ as it inherits its methods from its abstract Teleporter parent class. This also follows the open/closed principle, as therefore many different types of environments like the GoldenFogDoor may exist to move the players / actors between maps; however the Teleporter class is closed for modification.

Finally, the MapManager class was created to follow interface segregation and the single responsibility principle. Without this class, each map would have to be implemented within the ‘Application’ class in the main method; violating object oriented programming principles by modifying the previously existing code. As such, the MapManager class handles the constructors for each Map; namely the actual map strings to be printed and the constructors for the GoldenFogDoors which allow for seamless player movement between maps. Continually, the MapManager class follows object oriented principles by making use of the engine’s preexisting GameMap class and FancyGroundFactory class. These classes are both necessary in the printing of the map from Assignment 2; and have been implemented by the MapManager class to print the new maps Limgrave, Stormveil Castle, the Roundtable Hold and the Boss room.

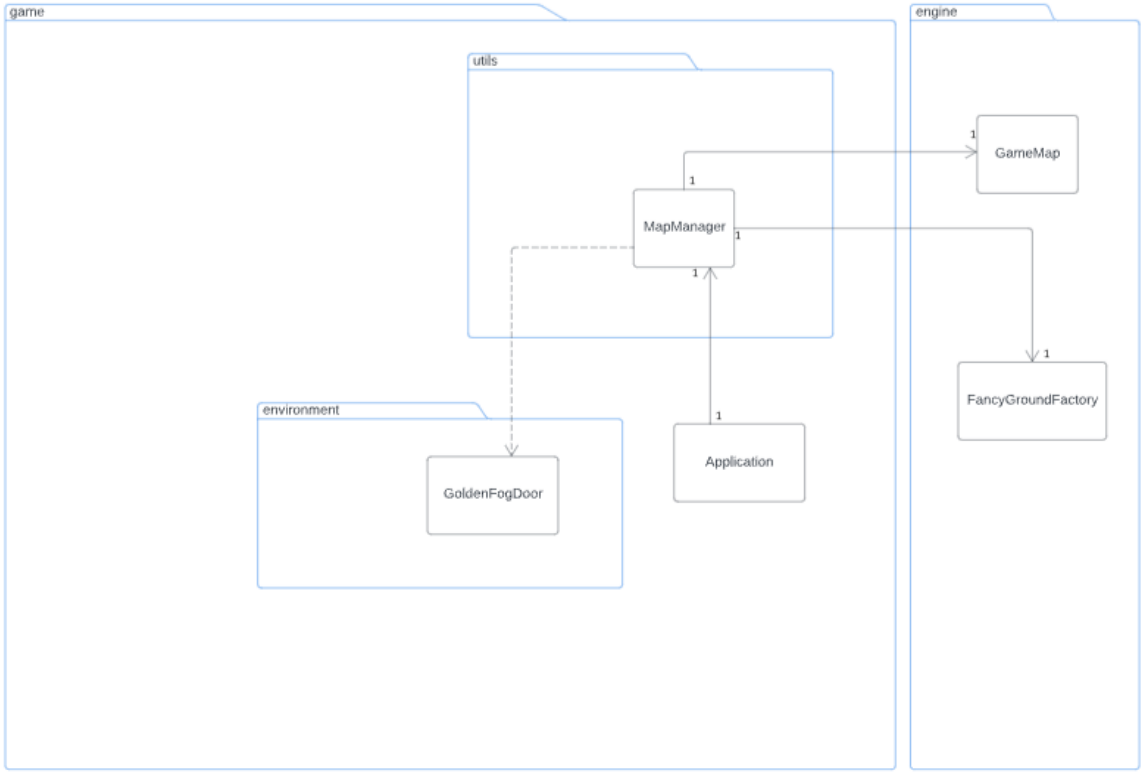
**Pros**
A benefit of Req 1's current implementation is that each new implementation has its own class and does not rely purely on modifying pre-existing classes, such as the FallAction being a child class of DeathAction, and the GoldenFogDoor being a child class of the abstract Teleporter class. This is beneficial as many more unique DeathAction child classes can exist via inheritance; which also applies to the Teleporter class.

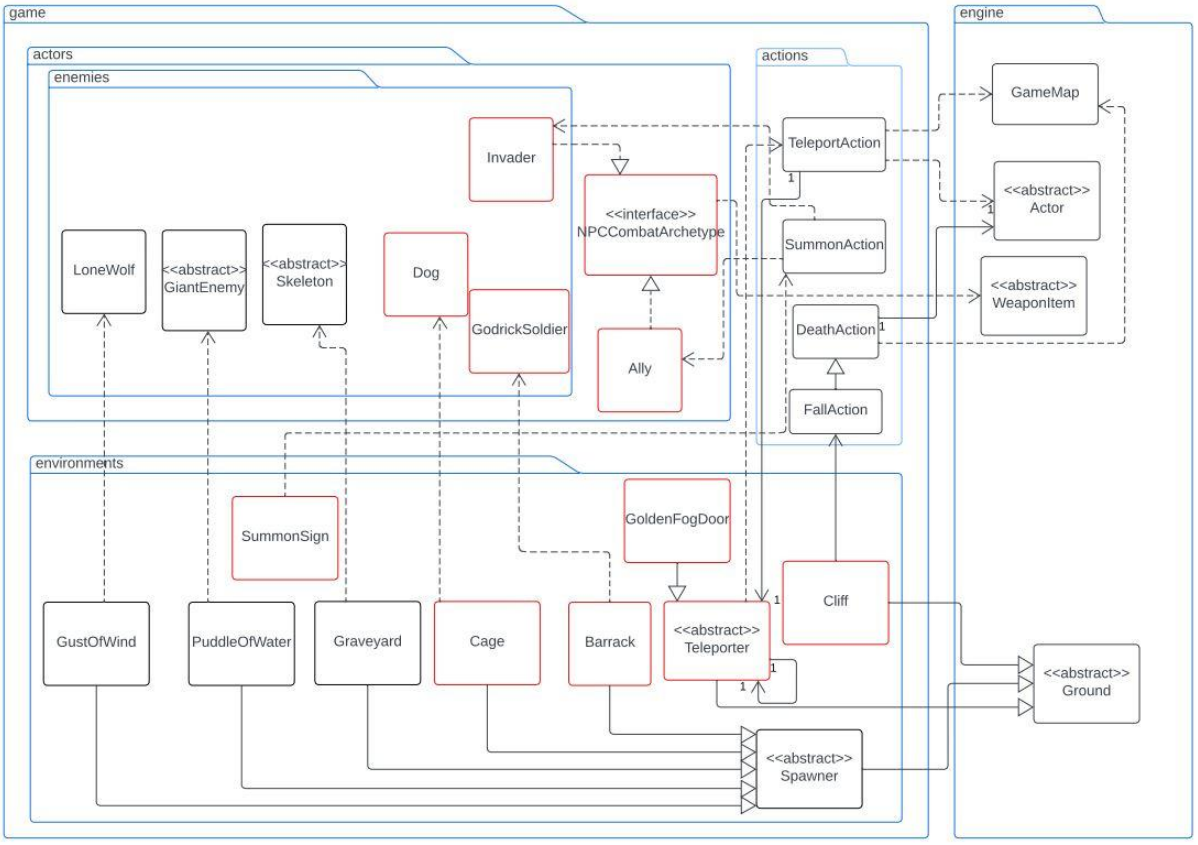
Continually, another benefit of this implementation is that the MapManager class makes full use of the engine's pre-existing code; specifically the FancyGroundFactory and GameMap class. This follows the DRY principle as if these classes were not used, an entirely new method of constructing and printing the ground would be a waste of time/effort and would lead to repetitive code and an inefficient use of space.

**Cons**
A negative aspect of this current implementation would be that each new map does not have its own class. If a new map is to be added in the future, it would have to be appended to the end of the MapManager class, thus violating the Open Closed principle.

Another negative aspect of this current implementation is that the MapManager class is also solely responsible for constructing the GoldenFogDoors for each respective map. This is reminiscent of the first negative aspect of this implementation, as if a new map is to be added, it will have to be appended to the end of the MapManager class; and as a byproduct of this implementation, the constructors for the new map's GoldenFogDoors will also have to be appended, violating the Open Closed principle.

# REQ 1 - UML DIAGRAMS



# REQ 2 - DESIGN RATIONALE

New Environment Classes

- Cage
- Barrack

New Enemy classes

- Godrick Soldier
- Dog

Modified classes from A2

- Abstract Enemy class
  - And as a result, all enemies (HeavySkeletalSwordsman, SkeletalBandit, GiantGrab, GiantCrayfish, GiantDog, LoneWolf)
- New Spawnable interface
- New 'STORMVEIL_FRIENDLY' status

The new environment classes, Cage and Barrack, both extend the pre-existing Spawner class from Assignment 2. This follows the Single Responsibility principle and the Open Closed principle, as similar to Assignment 2, these environments are solely responsible for spawning their respective enemies. They provide constructors for their enemies, define their displayChar and then, overriding the Tick method provided by the superclass Location, have a set chance to spawn their respective enemies per turn.

The new enemy classes, Godrick Soldier and Dog, both extend the pre-existing Enemy class from Assignment 2. This also follows the Single Responsibility principle and the Open Closed principle. Like any enemy from Assignment 2, these new enemy classes have been implemented in the exact same manner; they are solely responsible for setting their dropped runes (between 2 set integers), defining their special attacks, displayChar and any other parameters like health and damage. Continually, both the new Dog and Godrick Soldier, as a result of them both being raised / trained in the Stormveil Castle, do not attack each other. This is due to the new implementation of the 'STORMVEIL_FRIENDLY' status, which checks if both the target actor and attacking actor have this status. If both of these actors have the 'STORMVEIL_FRIENDLY' status, the attack will not be considered an allowable action.

Previously, all enemies inherited a method from the abstract enemy superclass called getSpawnChance - a method that allowed each child class of enemy to define their chance of spawning on the map with a singular integer. However, with the introduction of an enemy in Req 4 that has a completely different spawning method, it was decided to change the implementation of this getSpawnChance method. Continually, our feedback from Assignment 2 stated that 'the abstraction applied to enemies may cause sophistication in the future (spawning logic)'. Our interpretation of this feedback was that since the superclass Enemy is abstract, all child classes thus inherit all abstract methods, and this includes the getSpawnChance - and this could cause future complications with the introduction of the new 'Invader' enemy in Req 4, which has a unique spawning method. Thus, it was decided to remove this getSpawnChance method from the abstract Enemy class and reintroduce this method via an interface - called 'Spawnable'. All 'spawnable' enemies (enemies from Assignment 2 and the enemies in Req 2 of this assignment, Godrick Soldier and Dog) implement this interface as they all spawn from their own respective environments. This allows for enemies with simple or complex spawning logic. While we received feedback

regarding the IsWest method previously implemented in Assignment 2, we felt it was not necessary to update it as it was not relevant for Assignment 3.

**Pros**

A benefit of this implementation is that only the spawnable enemies implement the ‘Spawnable’ interface - allowing for more complex spawning logic with the implementation of ‘Invader’ in req 4, as well as in the future. This follows the Open Closed principle.

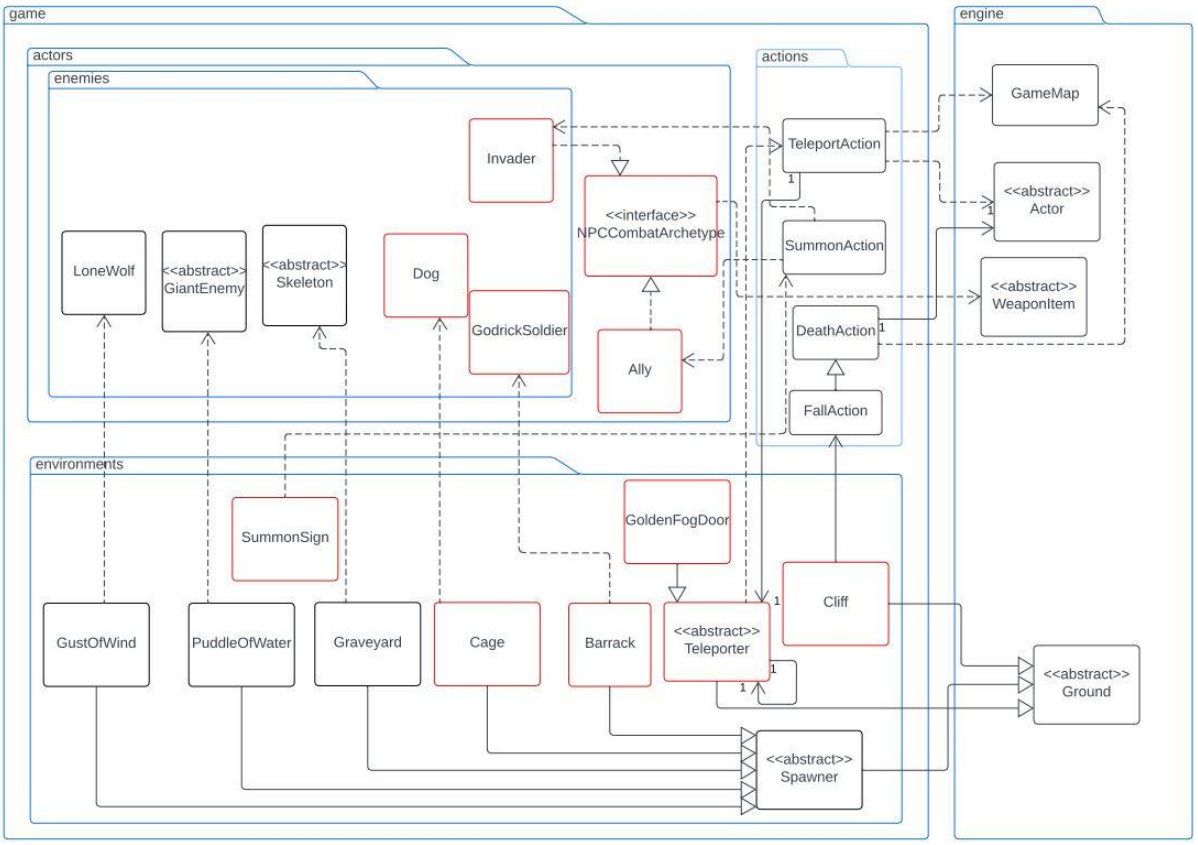
Another benefit of this current implementation is the further use of the environments and enemy blueprints from Assignment 2, following the DRY principle and by extension, the Open Closed principle and the Single Responsibility principle. The Barrack and Cage environments extend the abstract Spawner class, and the Dog and Godrick Soldier both extend the pre-existing Enemy class.

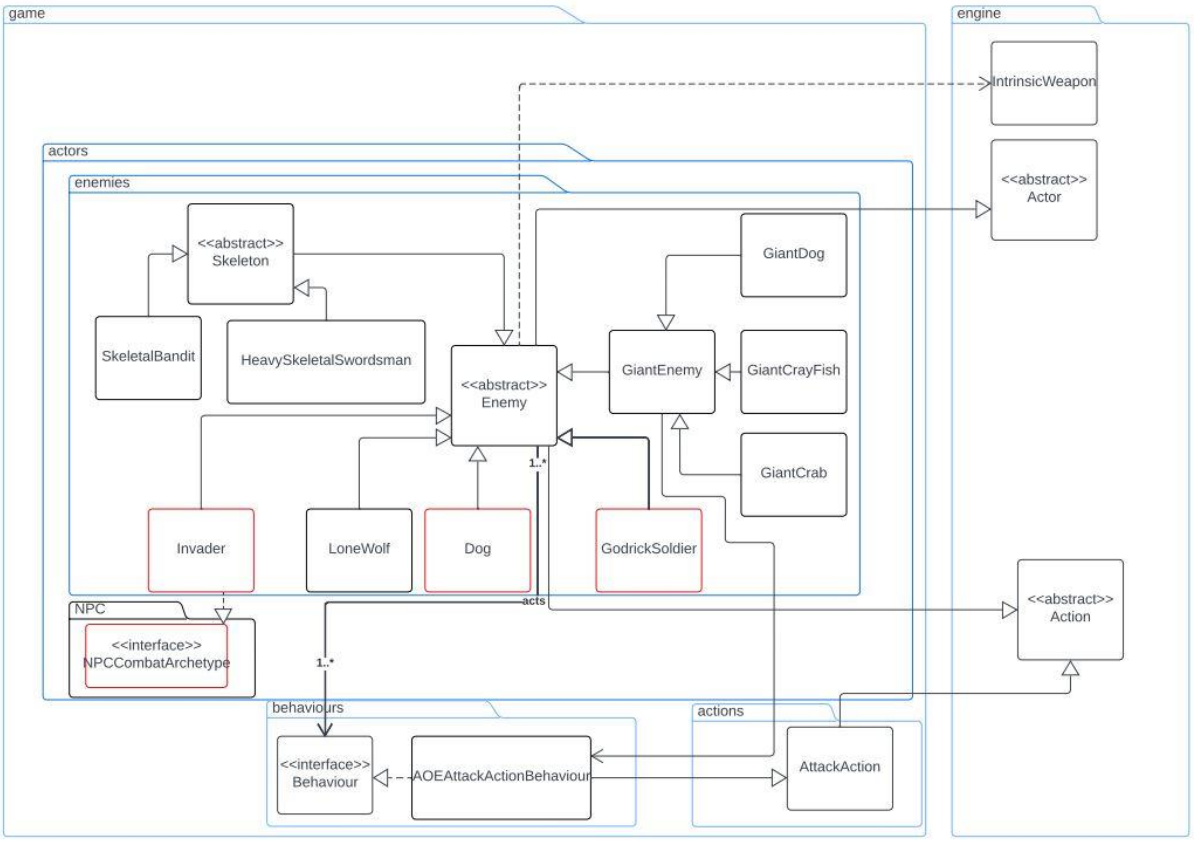
**Cons**

A negative aspect of Req 2’s implementation is that while the removal of the getSpawnChance method is productive, it may be tedious to implement the interface on each new enemy that is to be introduced with environment related RNG spawn logic. Thus, a fix for this issue could be to implement two different Enemy abstract classes - one with the getSpawnChance method, and one without.

Another negative aspect of Req 2’s implementation is that the implementation of the ‘STORMVEIL_FRIENDLY’ status was included by modifying the pre-existing BasicAttackAction, violating the Open Closed principle as it was added via modification and not extension.

# REQ 2 - UML DIAGRAMS



# REQ 3 - DESIGN RATIONALE

New classes in our implementation:

- Golden Runes
- Axe of Godrick
- Grafted Dragon
- Remembrance of the Grafted
- Finger Reader Enia

Modified classes from Assignment 2

- ConsumeHealAction
    - Following the feedback from Assignment 2, this has been further abstracted to create a ConsumeAction, which is used by the Golden Runes
- Selling items
    - Selling is now managed by the player as was suggested by the feedback for Assignment 2
- Buy price method
    - The Buy price method in all the weapons has been changed to get rid of returning negative numbers

The new Golden Runes item is scattered around the map and can be picked up by the player. The player can consume them using the new ConsumeAction and in turn will receive a random number of runes ranging from 200-10000. This item extends the Item class and implements a new Consumable interface, which is only used by Consumable items, ensuring that it follows the Interface Segregation Principles and furthermore by introducing abstraction in our Consuming, we follow the DRY principle too.

Two new weapons were introduced; the Axe of Godrick and the Grafted Dragon. Both of these weapons inherited from the WeaponItem class, as previous weapons from Assignment 2 did and both implement the Sellable interface, but not the Buyable. The use of separate interfaces ensures that we support the Interface Segregation Principle. Concerning weapons, the QuickStep and Unsheathe methods, while suggested to be changed in the Assignment 2 feedback, were left the same due to time constraints and these actions had no relation to Assignment 3.

The new Remembrance of the Grafted item inherits the Item abstract class and also the Sellable interface. It can either be sold to any merchant for 20000 runes or can be traded to Finger Reader Enia for either the Grafted Dragon or the Axe of Godrick. This once again utilises the Interface Segregation Principle, as this item is not Buyable, only Sellable. This item should be dropped from Godrick the Grafted, however due to the optional nature of this requirement and due to time constraints, this was not implemented. Additionally, the feedback from Assignment 2 regarding the Site of Lost Grace was not implemented either due to not having relevance to Assignment 3 and due to time constraints.

The new NPC, Finger Reader Enia, is capable of buying all Buyable Weapons from the player but can also trade the Remembrance of the Grafted. In accordance with the feedback from Assignment 2, as a Merchant/Trader, Enia and Merchant Kale are no longer responsible for buying items from the player, and this now upholds the Single Responsibility

Principle. Selling items to merchants is now managed by the Player. Furthermore, string checking for weapons to sell has been removed to take into consideration our feedback from assignment 2. The return weapon item method has similarly been removed to adhere to the Assignment 2 suggestions.

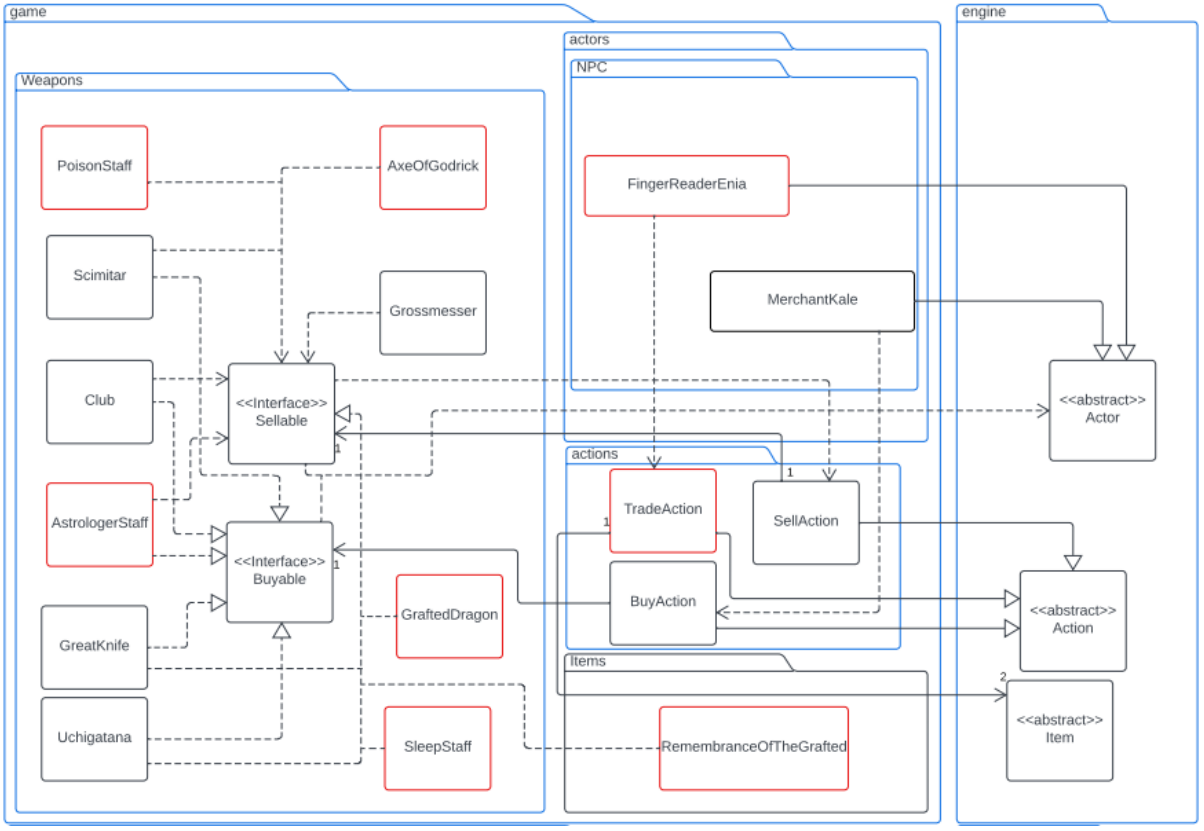
A benefit of our design choice is the decision to change the Player to manage their own sellable items. This upholds SRP with both Merchant Kale and Finger Reader Enia, as they are just responsible for selling items to the player and in Finger Reader Enia's case, trading weapons and items with the player.

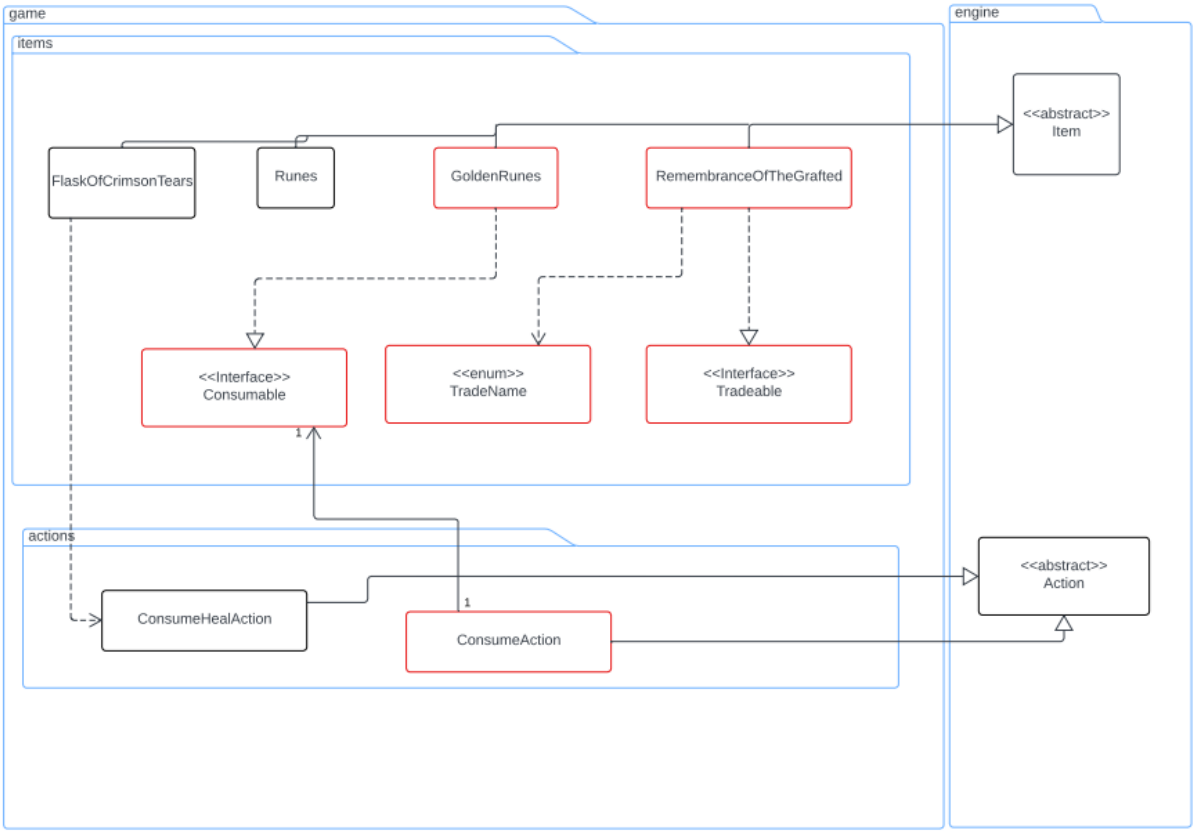
A pro of our design is that by separating the Sellable and Buyable interfaces, we uphold the Interface Segregation Principle which states that classes should not be forced to depend on interfaces that they do not use. In the case of our game, not every weapon is buyable and sellable and so by having 2 different interfaces, we ensure that we can select which weapons we want to be buyable and which we want to be sellable.

A con of our implementation could be that the Finger Reader Enia and Merchant Kale do not both inherit from a Merchant or Trader class despite them both being interactable NPCs. However our decision to not have them be child classes of a Merchant class was because while they are both interactable, their functions are different. They can both buy weapons from the player however this is managed in the player class. Apart from this, Merchant Kale can sell weapons to the player which Enia cannot do, and Enia can trade weapons and items with the player, which Kale cannot do.

Another con of our design could be that all sellable weapons need to have the code to allow the weapon to be sellable and is not in an abstract class. This is because not all weapons are sellable so cannot inherit from a sellable class but the sellable weapons can implement a sellable interface.

# REQ 3 - UML DIAGRAMS



# REQ 4 - DESIGN RATIONALE

New classes in our implementation:

- Astrologer
- Astrologer's Staff
- Summon Sign
- Summon Action
- Ally
- Invader

Modified classes from A2:

- Combat Archetypes
    - Instead of the combat archetypes inheriting the player class, we utilised composition and had the combat archetypes inherit from their own class which were then contained in the player
- Player Selection Manager
    - Used to select the players combat archetype which was then called in Application.java

The new Astrologer class implemented the new Combat Archetypes class similar to the previous combat archetypes from Assignment 2. Following the feedback from Assignment 2, composition was used for the players classes rather than inheritance. The Astrologer Staff, a new weapon, inherited the existing WeaponItem class and utilises the existing Buyable interface used for trading. The combat archetype class that was implemented upholds the Open Closed Principle with the use of its abstraction.

The two new NPCs, Ally and Invader were both able to use the combat archetypes of the player, with their respective weapons and hitpoints. The use of the new Combat Archetypes class allowed us to have a class randomiser method to adhere to the requirements of task 4 and apply the hitpoints and weapon of the random class to the new Ally or Invader as they spawn. The Invader inherits from the existing Enemy class, following the Single Responsibility Principle as previous enemies were in Assignment 2. It is responsible for its unique number of runes, its Display Char and other parameters such as health and damage. The Ally on the other hand inherits from Actor, again following SRP where it is responsible for its own unique characteristics. Additionally, both of these classes adhere to the Open Closed Principle too. Furthermore, these NPCs have their own Status to check their hostility to other mobs and depending on this, will attack their enemies but not their friendly entities.

Both of these NPCs are summoned using the Summon Sign, an environment class that inherits from the Ground. This Ground allows the user to perform a Summon Action when standing next to it, which will spawn either an Ally or an Invader with a 50% chance for each. The Summon Action extends the existing Action class and uses the execute method to check for a valid spawning location in the Summon Signs exits. The Summon Sign ground follows SRP as this ground type's only purpose is to summon either Allies or Invaders.

A benefit of our current implementation is that the Ally and Invader classes can utilise the abstract Combat Archetypes class to get a random class. Without this implementation, either multiple if statements or perhaps for loops would need to be used to get the weapons and hit

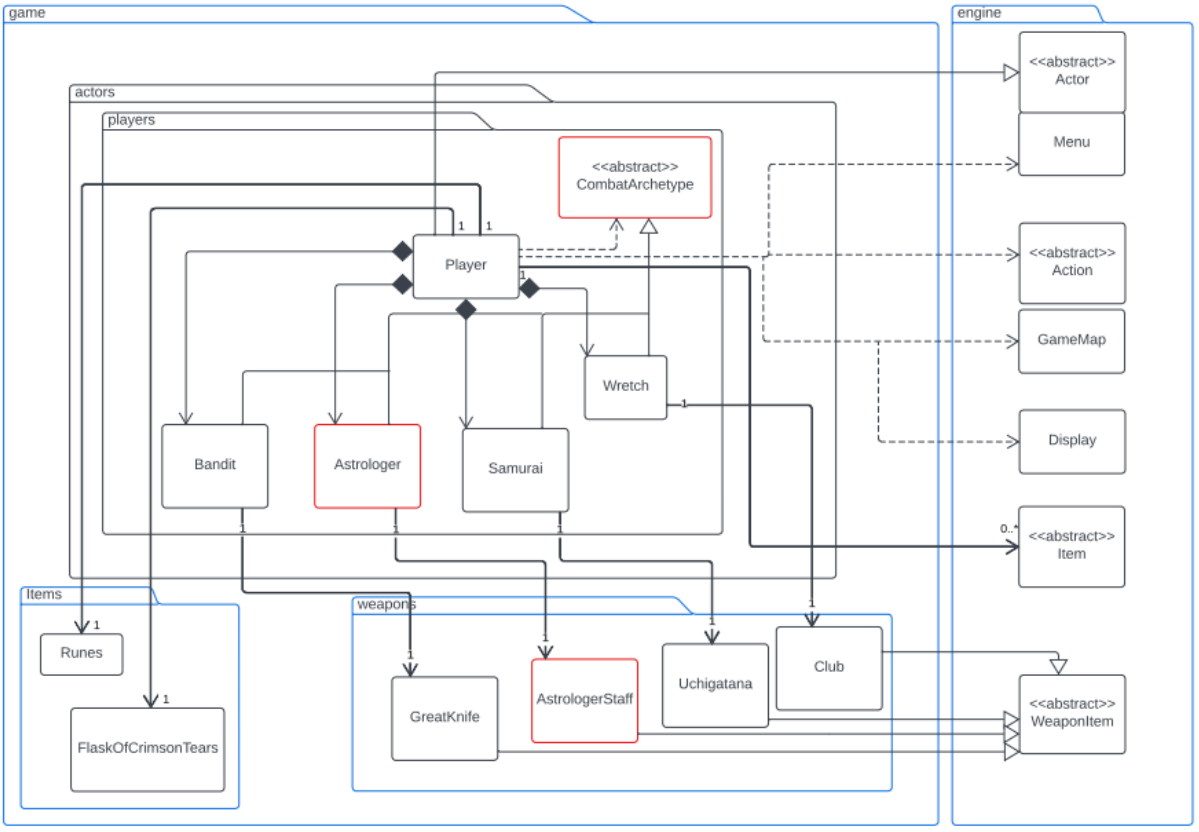
points from the old combat archetype implementation before applying them to these new NPCs to spawn.

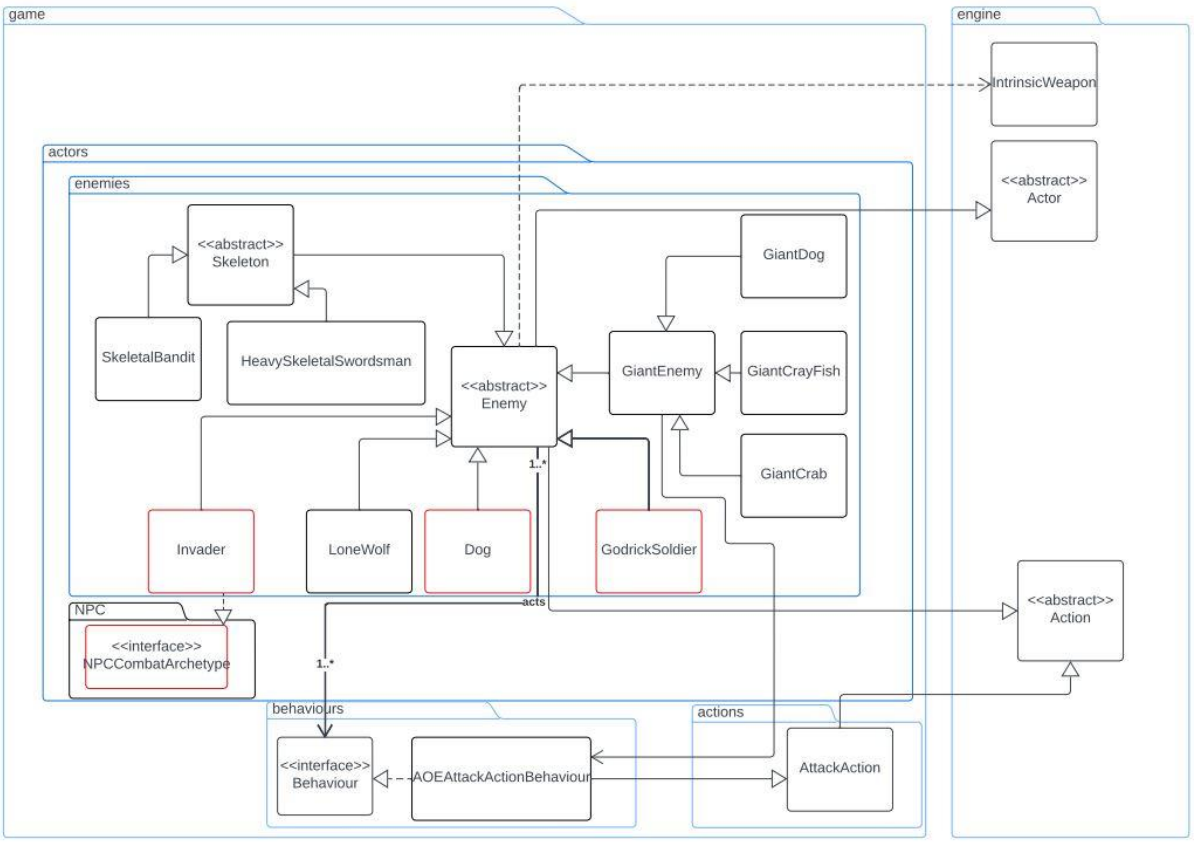
Another benefit of our design choices is that we apply the use of the Single Responsibility Principle well with our spawning logic for the Summon Sign. The Summon Sign calls its own Summon Action to spawn either an Ally or Invader, as these are unique to the Summon Sign, rather than calling a Spawner class.

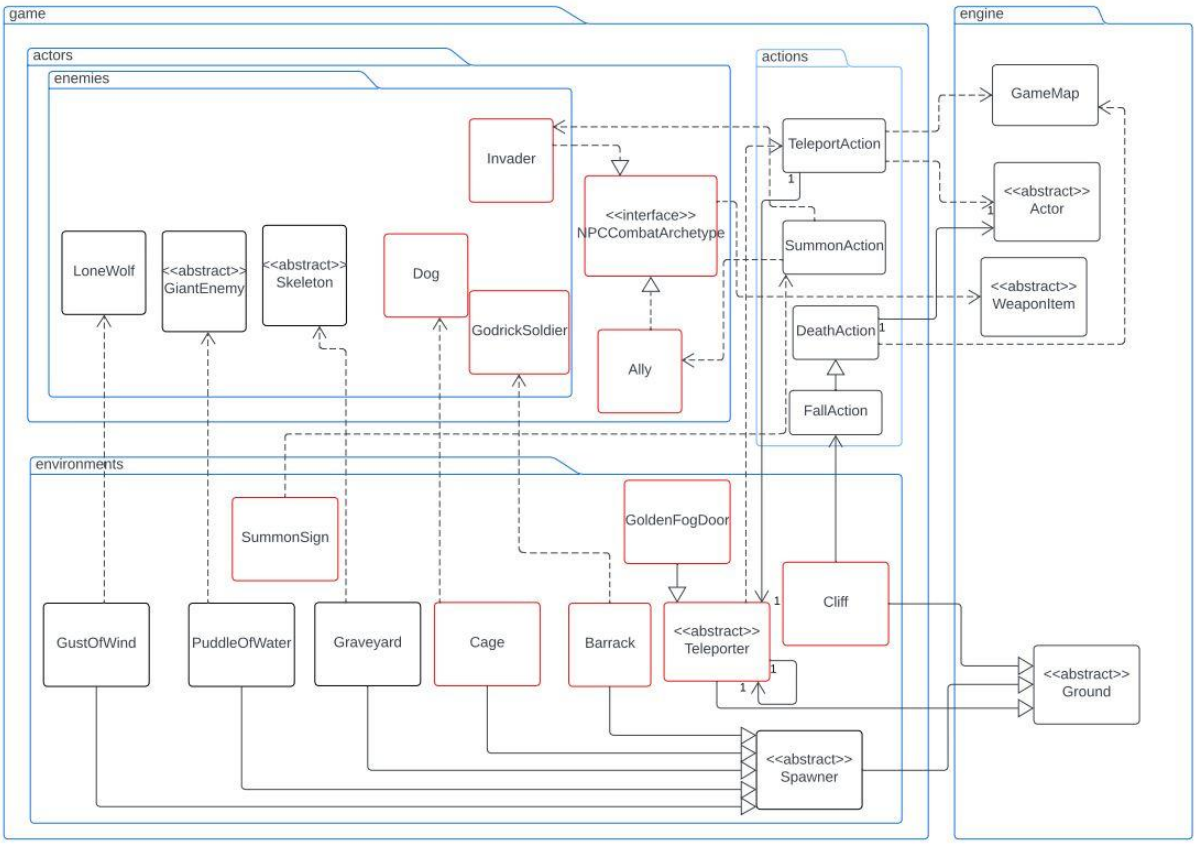
A possible flaw of the current design could be with each new entity added, a new if statement has to be introduced in the BasicAttackActionBehaviour class to check whether an entity should be able to attack its own kind or not.

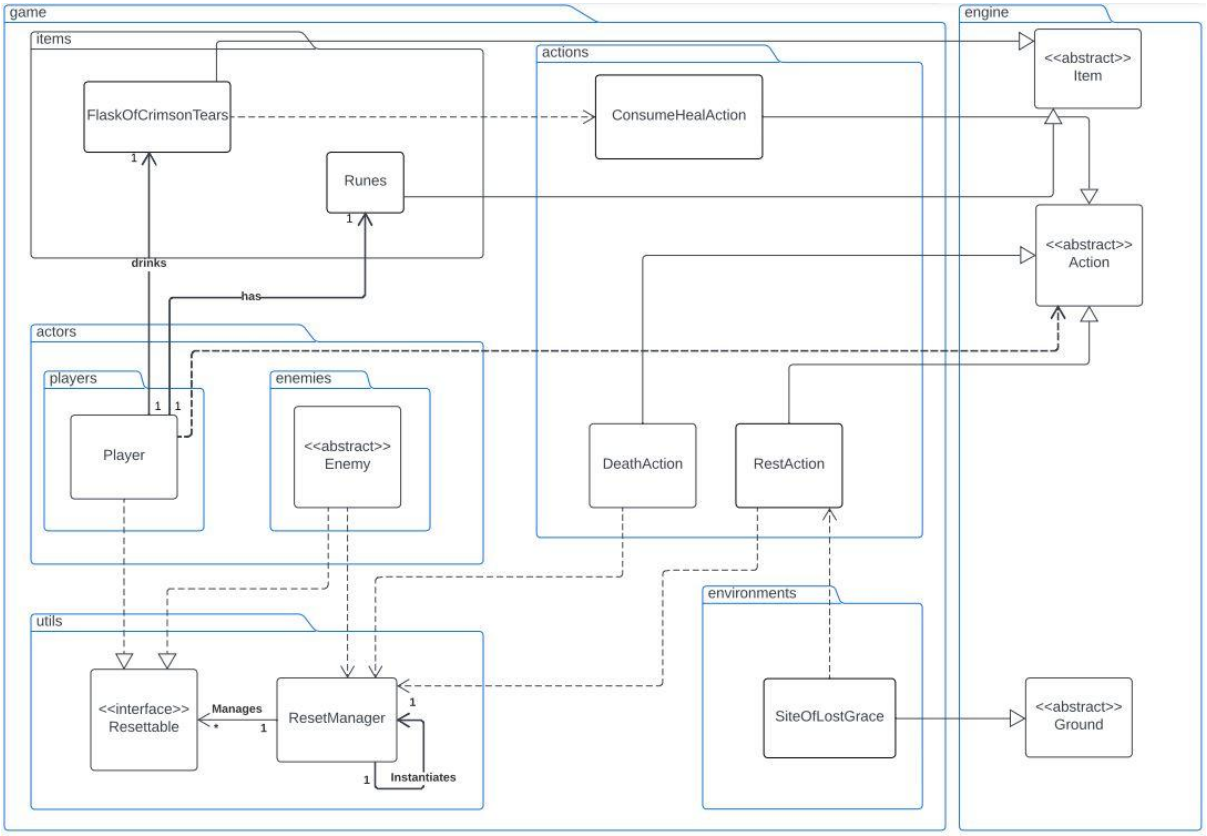
Another possible flaw could be that the BasicAttackActionBehaviour uses an intrinsic weapon by default to attack but the Ally and Invader both use weapons. The Invader had to have the BasicAttackActionBehaviour from the parent Enemy class removed and added back to use its weapon.

# REQ 4 - UML DIAGRAMS




game
engine
IntrinsicWeapon
actors
enemies
<<abstract>>
Skeleton
<<abstract>>
Actor
GiantDog
SkeletalBandit
HeavySkeletalSwordsman
<<abstract>>
Enemy
GiantEnemy
GiantCrayFish
GiantCrab
1..*
Invader
LoneWolf
Dog
GodrickSoldier
acts
<<abstract>>
Action
NPC
<<interface>>
NPCCombatArchetype
1..*
behaviours
actions
<<interface>>
Behaviour
AOEAttackActionBehaviour
AttackAction


game
engine
actors
enemies
actions
GameMap
Invader
TeleportAction
1
<<abstract>>
Actor
<<interface>>
NPCCombatArchetype
SummonAction
LoneWolf
<<abstract>>
GiantEnemy
<<abstract>>
Skeleton
Dog
GodrickSoldier
<<abstract>>
WeaponItem
DeathAction
1
Ally
FallAction
environments
GoldenFogDoor
SummonSign
Cliff
GustOfWind
PuddleOfWater
Graveyard
Cage
Barrack
<<abstract>>
Teleporter
1
1
1
<<abstract>>
Ground
<<abstract>>
Spawner




# REQ 5 - DESIGN RATIONALE

New Classes

- IsAsleepAction
- CastSleepAction
- PoisonAction
- Abstract DelayedDamageWeapon
    - SleepStaff
    - PoisonStaff

Req 5 of this assignment was a creative requirement - as a team, we decided to implement new weapons with status effects - specifically a PoisonStaff (which poisons an enemy for 3 turns) and a SleepStaff (which renders an enemy asleep and unable to perform an action for 1 turn).

This new requirement uses two classes from the engine package; specifically Action and WeaponItem. IsAsleepAction, CastSleepAction and Poison action all extend AttackAction which extends the abstract Action class. Continually, the WeaponItem class is extended by both the SleepStaff and PoisonStaff.

It is compulsory that this creative requirement also reuses at least 1 existing feature from Assignment 2 or a fixed requirement from Assignment 3 - both the PoisonStaff and SleepStaff are buyable and sellable with MerchantKale, a previously existing feature from Assignment 2. The new requirement must use existing or create new abstractions - the DelayedDamageWeapon (which the poisonstaff extends) is abstract. The new requirement

must use existing or create new capabilities - this was fulfilled with the creation of the new ‘POISONED’ and ‘ASLEEP’ statuses.

The IsAsleepAction, CastAsleepAction and PoisonAction all extend AttackAction, following the DRY principle. Continually all these actions follow the Single Responsibility Principle as they are all solely responsible for their unique attack action’s details, just like every other action. This aligns with the Interface Segregation principle, as each unique attack is separated into its own class. This also follows the dependency inversion principle, as the Action superclass is abstract.

The abstract DelayedDamageWeapon follows SRP as it is solely responsible for defining a weapon’s damage over time. It also extends the abstract WeaponItem class, following the DRY principle. As both of these classes are abstract, this is also in line with the Dependency Inversion principle. Continually, this fits Interface Segregation, as weapons that do not deal damage over time can extend WeaponItem, and vice versa. This follows the Open Closed principle as the previously existing WeaponItem class (provided by the game engine) was extended to this new child class with new code, without the previously existing code to be modified / affected.

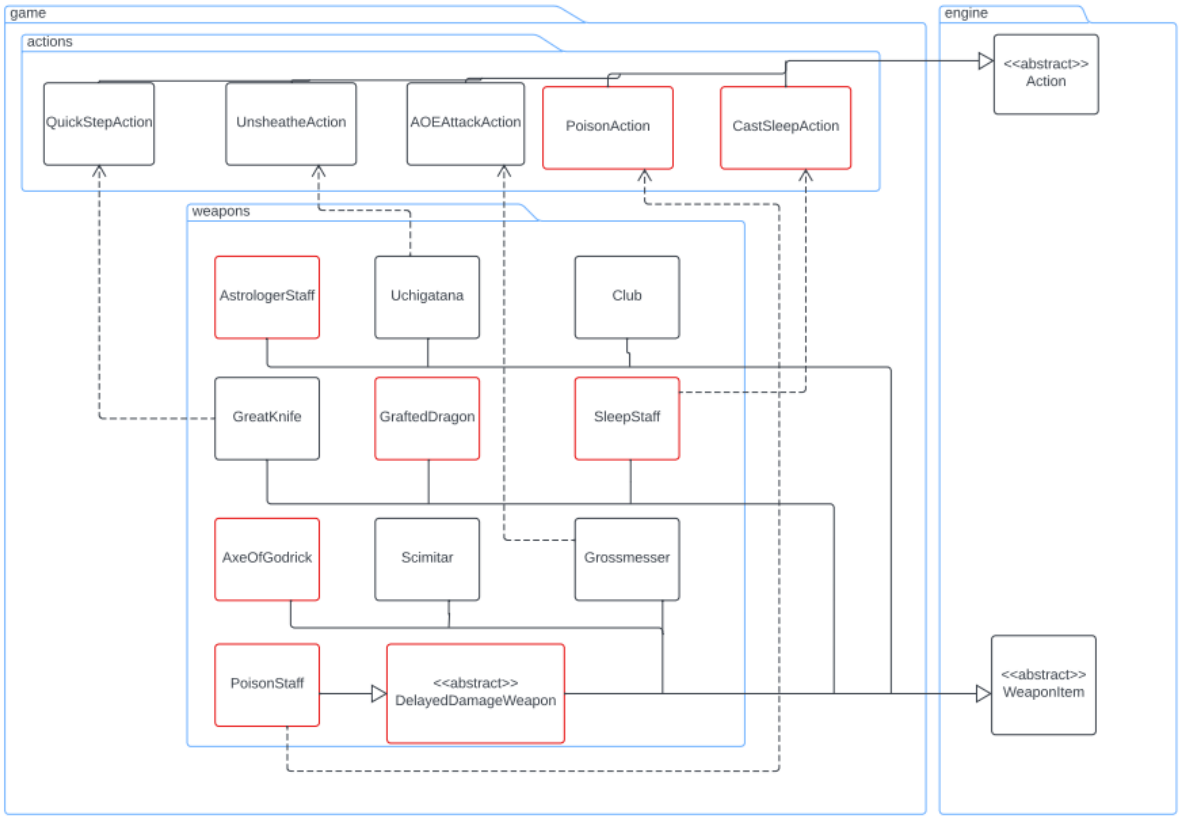
**Pros**
A benefit of this implementation is that this creative requirement does not affect any of the previously existing code - every new class in this implementation is entirely new and does not impact the implementation of Assignment 2 (and even works with it, as the new weapons can be bought and sold by Merchant Kale!), following the Open Closed Principle.

Another benefit of this implementation is the use of the abstract DelayedDamageWeapon class - alternate damage over time weapons can be easily implemented with this abstract class in the future, if need be - following the DRY principle and the Dependency inversion principle.

**Cons**
A negative aspect of this implementation is that the IsAsleep action only lasts for 1 turn. This is a flaw as in the future, a new Sleep Item may be added that makes an actor sleep for more than 1 turn. Modifying this code will thus violate the Open Closed principle.

Another negative aspect of this implementation is that the PoisonAction lasts for 3 turns - similar to the implementation of the IsAsleep action. If a future weapon is to be implemented in which the number of turns the poison lasts is different, this will need to be changed and the implementation of a new poison weapon will thus violate the Open Closed principle.

# REQ 5 - UML DIAGRAMS




# CONTRIBUTION LOG

https://docs.google.com/spreadsheets/d/1ESvnFQNk3XJ2QLDXeMLZOv2KC0u1jdanL7zptNXUyIY/edit#gid=846598609