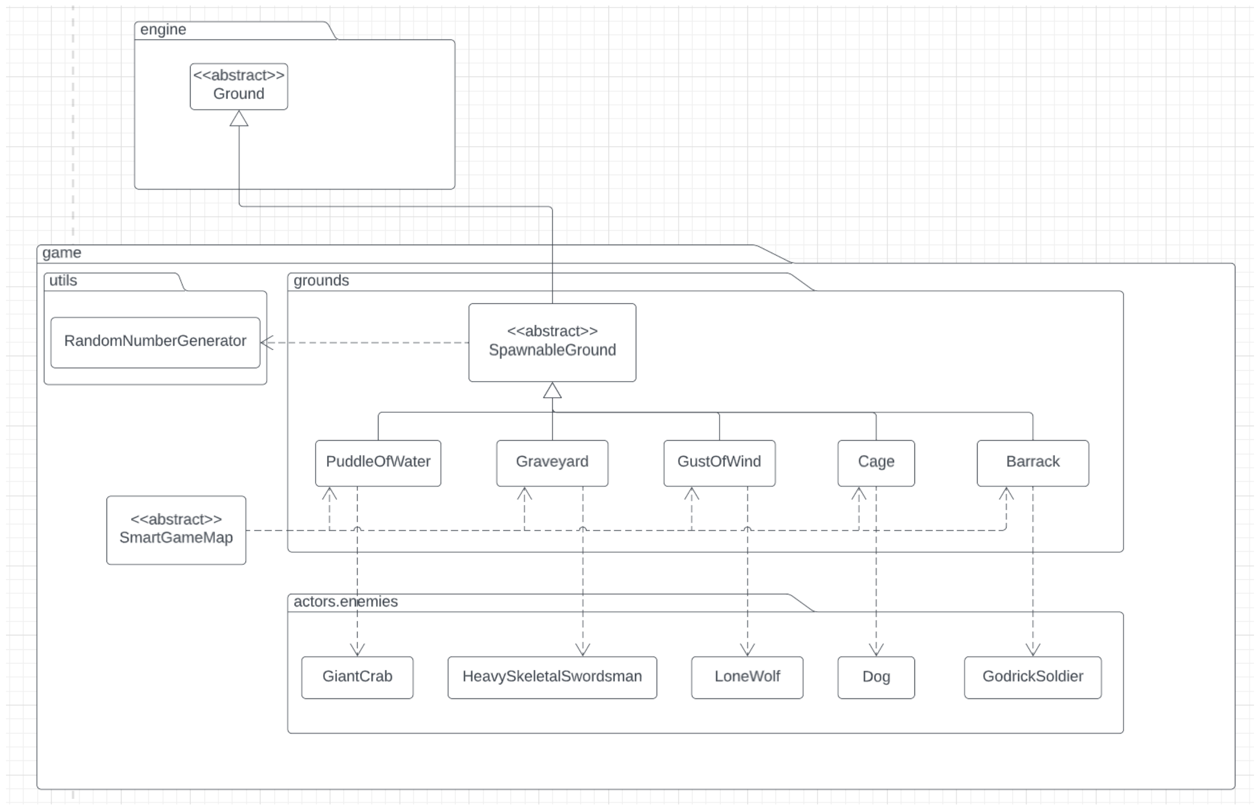
Design Rationale Assignment 1,2,3

William, Salar and Ibrahem

REQ1A (Updated)



The figure in REQ1A outlines a few new classes to extend the existing system:

- New Environments
    - Puddle of Water
    - Graveyard
    - Gust of Wind
- New enemies (Will be further discussed in REQ1B)

New additions as of Assignment 3

- Cage
- Barrack

The 3 new Environments will extend existing ground functionality similar to Floor, Wall and Dirt. This will allow the engine to render them for us using the existing code. To facilitate this, we will be making use of a “SpawnableGround” abstract class to allow us to reduce repetition in the code.

This will also minimize coupling in the event that we add more types of spawable ground tiles.

Inheriting ground functionality will allow the existing engine to interact with our new code without modification, hence following the open-closed principle.

As per the design criteria, the new ground extensions will be responsible for spawning the respective enemy to the GameMap. Hence each new child of ground will be dependent on the respective enemy type.

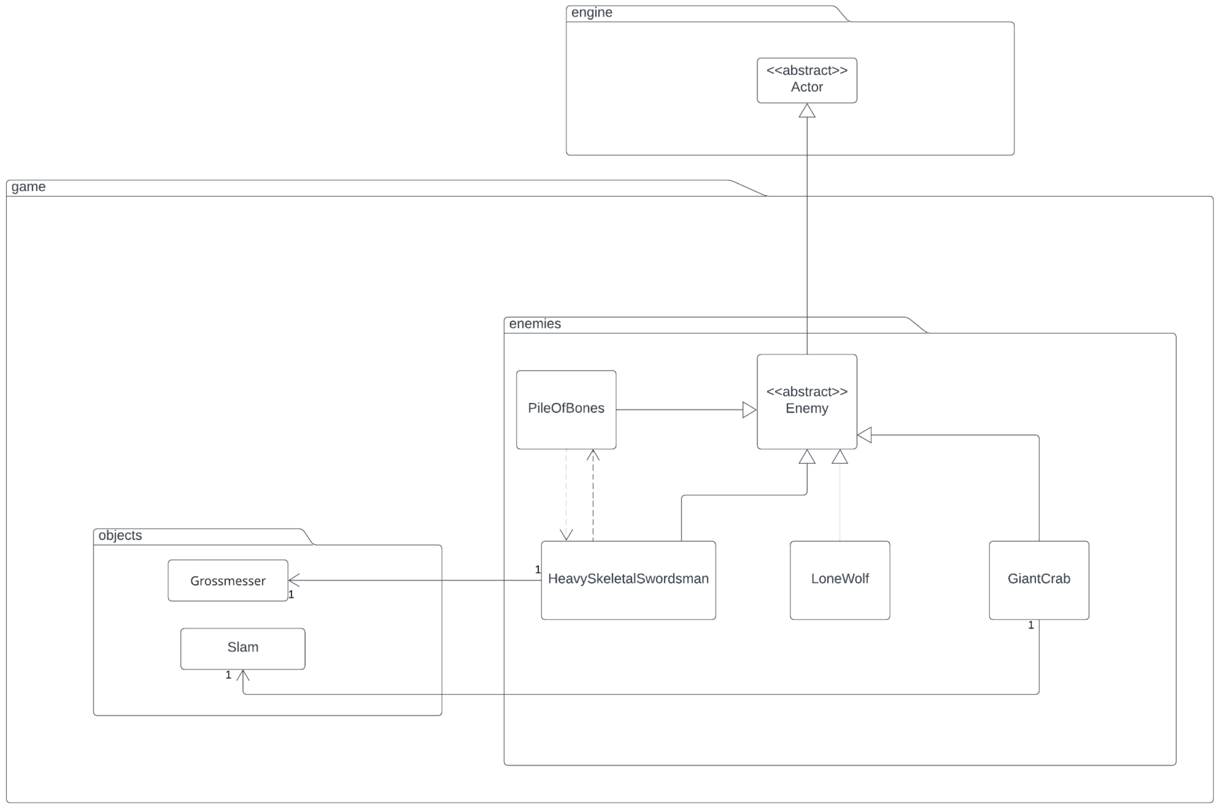
The 2 **new Assignment 3 grounds** can be seamlessly implemented within the established design as extensions of the SpawnableGround class.

Here is an outline of some alternative approaches and the associated pros and cons:

- Use of an interface to indicate that the ground is spawnable:
    - Pros:
        - Highlights that the classes that implement it are spawn tiles.
        - Allows additional functionality to be added to the spawn tiles in future.
    - Cons:
        - Redundant under the current requirements due to the already sufficient presence of a global clock method in the engine.
        - No default methods are reasonable for the spawnable ground classes.

- Removal of the abstract class:
    - Pros:
        - Makes it easier to program
    - Cons:
        - Potentially could cause WET issues if we were to add more functionality later on.
        - Increases coupling.

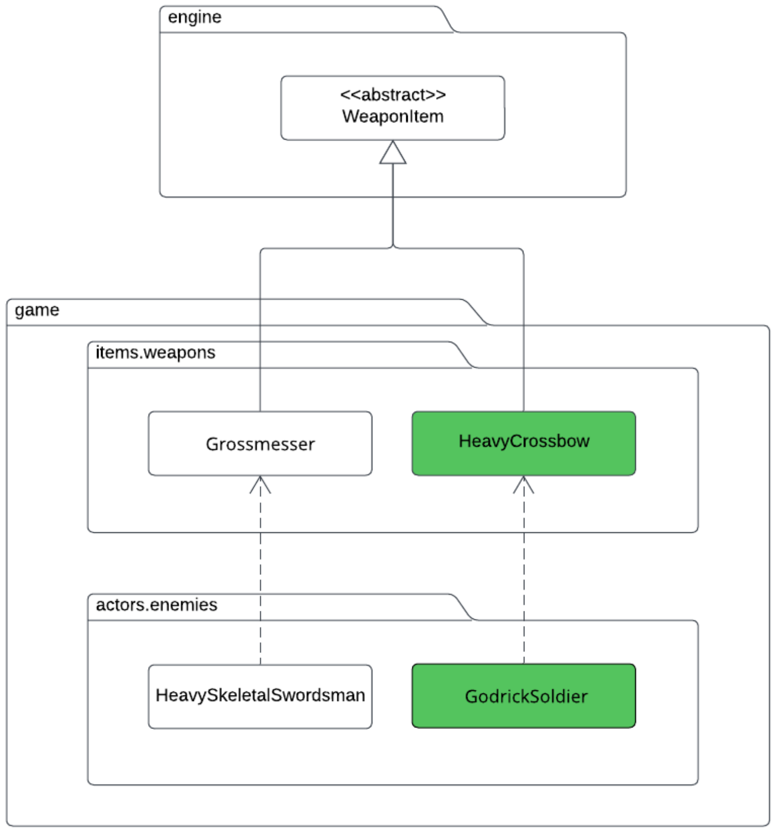
REQ1B



This requirement introduces 3 new enemies and their abilities.

- The heavy skeletal swordsman's Pile Of Bones was made another enemy as it needs the ability to be attacked so it will just be an enemy with no wander behavior and no attack behavior. Allowing it to be attacked but not attacked. Although this breaks ISP as there will be redundant methods. It is the best way as there is a do nothing attack action.
- Giant crab's intrinsic weapon will be made an actual weapon, allowing us to make the area slam a special attack instead of increasing the burden of giving an intrinsic weapon a special attack. This conforms to ISP as adding additional functionality to IntrinsicWeapon will be redundant to all other intrinsic weapons.
- A new Enemy superclass is made to direct responsibility for enemies onto a single actor class following SRP. Having a clear superclass for Enemy also follows the DRY principle.

REQ1C



-

This figure in REQ1C shows the following new classes:

- GodrickSoldier
- HeavyCrossbow

REQ1C

-

Weapons extends the WeaponItem abstract class since they are a kind of weaponItem and need to have all the attributes and methods that a WeaponItem has. (DRY)

No severe modification has been done inside the weapon classes and since they are weapons they can do anything that a weaponItem does aswell. (LSP)

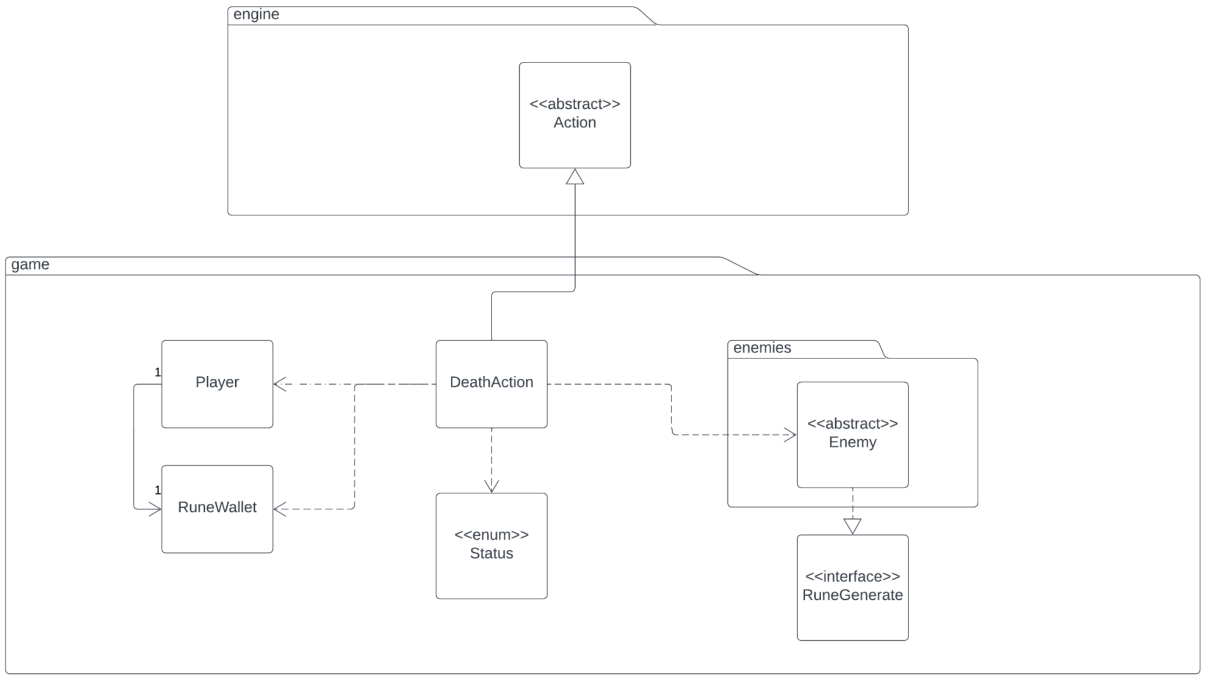
Enemies Create their own weapons.In this way we alleviate the responsibility of creating from the Application, so that Application would not have additional things to handle(SRP)

Other design implantation choices:

- Weapons are written from scratch:

  - Pros : each weapon can be build so that it behaves exactly as intended with no extra or redundant methods
  - Cons : writing a lot of repetitive code
- Weapons are initialised in application and added to enemies:
  - Pros : less dependency between enemy and weapon
  - Cons : application would become a godclass and there would be lot of coupling between application, weapon and enemy classes

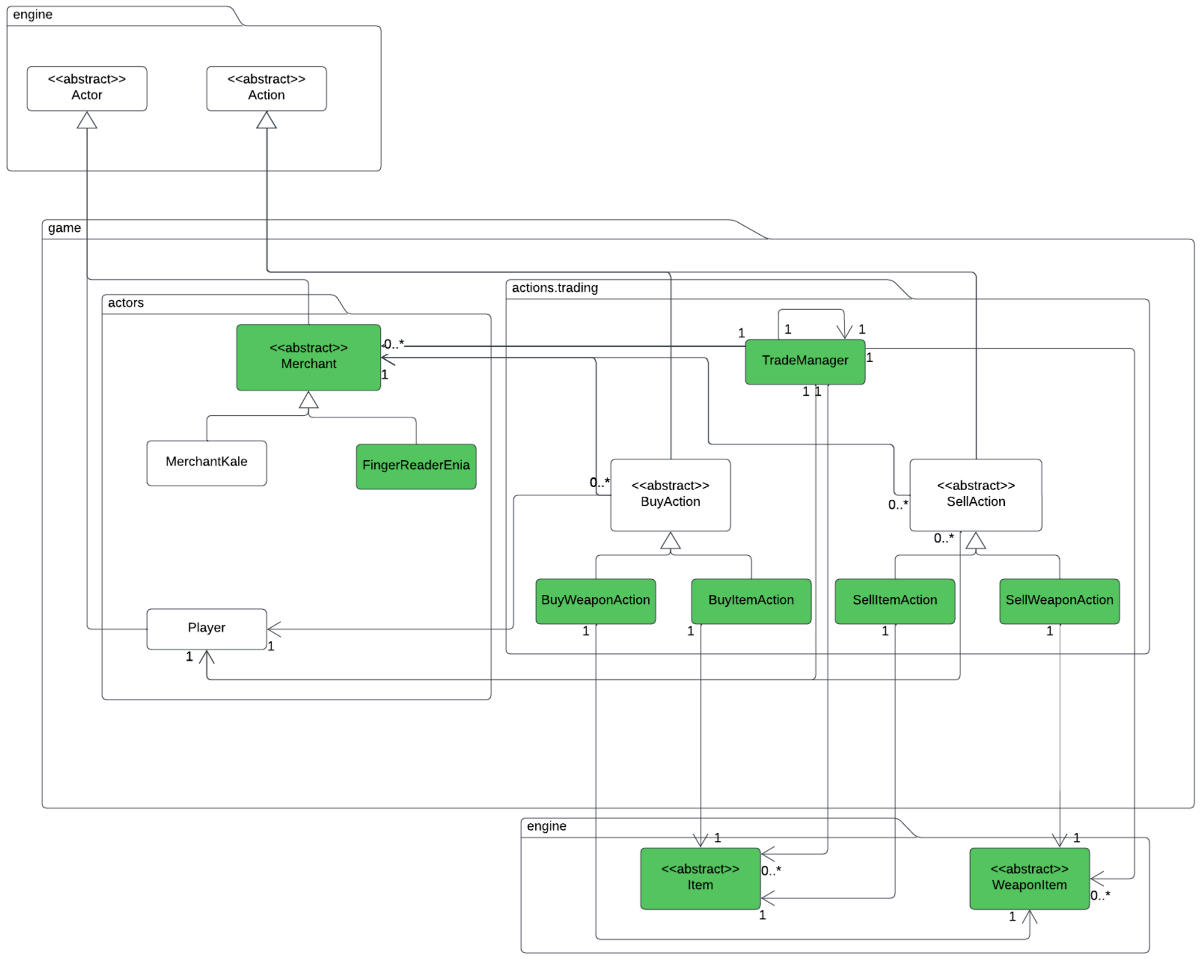
REQ2A



This requirement introduces the implementation of runes and enemy drop rates.

- RuneWallet will be used as an attribute for players. It should introduce an add rune, subtract rune, and drop runes (for death) functions focusing all the rune functionality on RuneWallet. Conforms with SRP as the functionality is separated from Player
- The RuneGenerate interface introduces an abstract method for generating runes for the player to add in their wallet. This means that any class that needs to generate runes can do so through this interface in the future giving sole responsibility for rune generation to the interface (SRP)
- The DeathAction class is the intermediary between the rune generation and RuneWallet. Although this creates a kind of mini god class and isn't really good the decision to give runes or not is decided through death action so i couldn't think of a better way to do it.

REQ2B



This figure in REQ2B shows the following new classes:

- TradeManager
- Item and WeaponItem
- Merchant
- FingerReaderEnia
- BuyWeaponAction and BuyItemAction
- SellWeaponAction and SellItemAction

MerchantKale and FingerReaderEnia extend Merchant. Since they need to use necessary methods provided by Merchant like the overridden Playturn(Dry). It also reduces added dependencies since Merchant is already dependant on them e.g. TradeManager. (ReD)

Merchant extends the actor class. since it needs to use the necessary methods provided by 'Actor' to handle its AllowableAction, Location, etc. (DRY)

The same logic is followed by BuyAction, sellAction and their children.

This design establishes a singleton pattern for the TradeManager class, allowing only one instance to exist at any given time. The TradeManager class initializes itself and stores the initial instance it creates. Any subsequent attempts to create new instances will return the previously created instance. This design ensures a solitary entity handles buying and selling operations, preventing potential issues that could arise from multiple TradeManagers coexisting during trades.

The TradeManager class follows the Single Responsibility Principle (SRP) by handling trade-related operations and ensuring a single instance exists at any given time

This feature is extensible because if any new Items, weapons or merchants need to be added they will just need to register with trademanager without needing to modify the class.

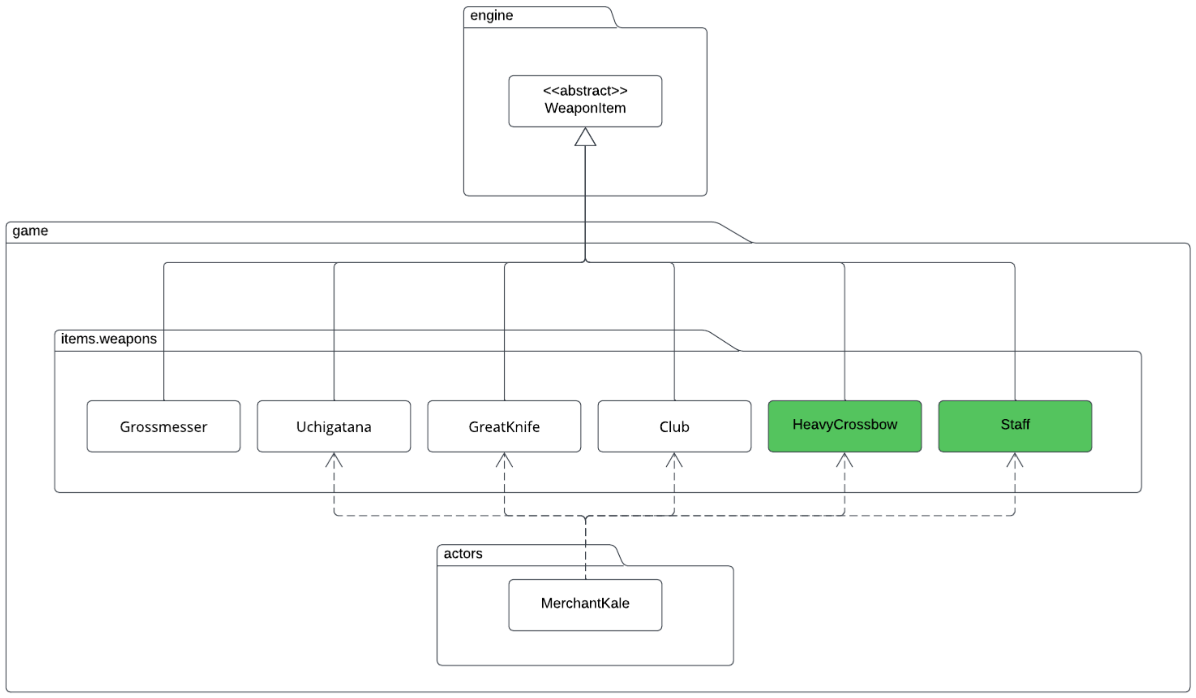
Other design implantation choices:

- Merchants Creating the BuyAction and SellActions:
    - Pros: no need for a manager, reduces coupling
    - Cons: No way of knowing if the Item is valid to buy from player without using an interface
- Using an Enum to check who is interacting with Merchant:
    - Pros : it would allow us to put the trader outside of floors in the future if needed
    - Cons :having extra Enums for a check that is not currently needed.
- Items keep track of their prices instead of TradeManager and merchants

    Pros : reduces additional coupling between tradeManager and merchants

    Cons : not very extensible if different merchants have different buy and sell prices

REQ2C



This figure in REQ2C shows the following new classes and interface:

- HeavyCrossbow
- Staff

Merchants create Weapons. Because we need to check what the merchants can sell the merchant needs to create the weapons itself. It really doesn't matter if the application makes the weapon or the merchant but since we don't want the application to handle too many things we just leave the responsibility to the merchant.

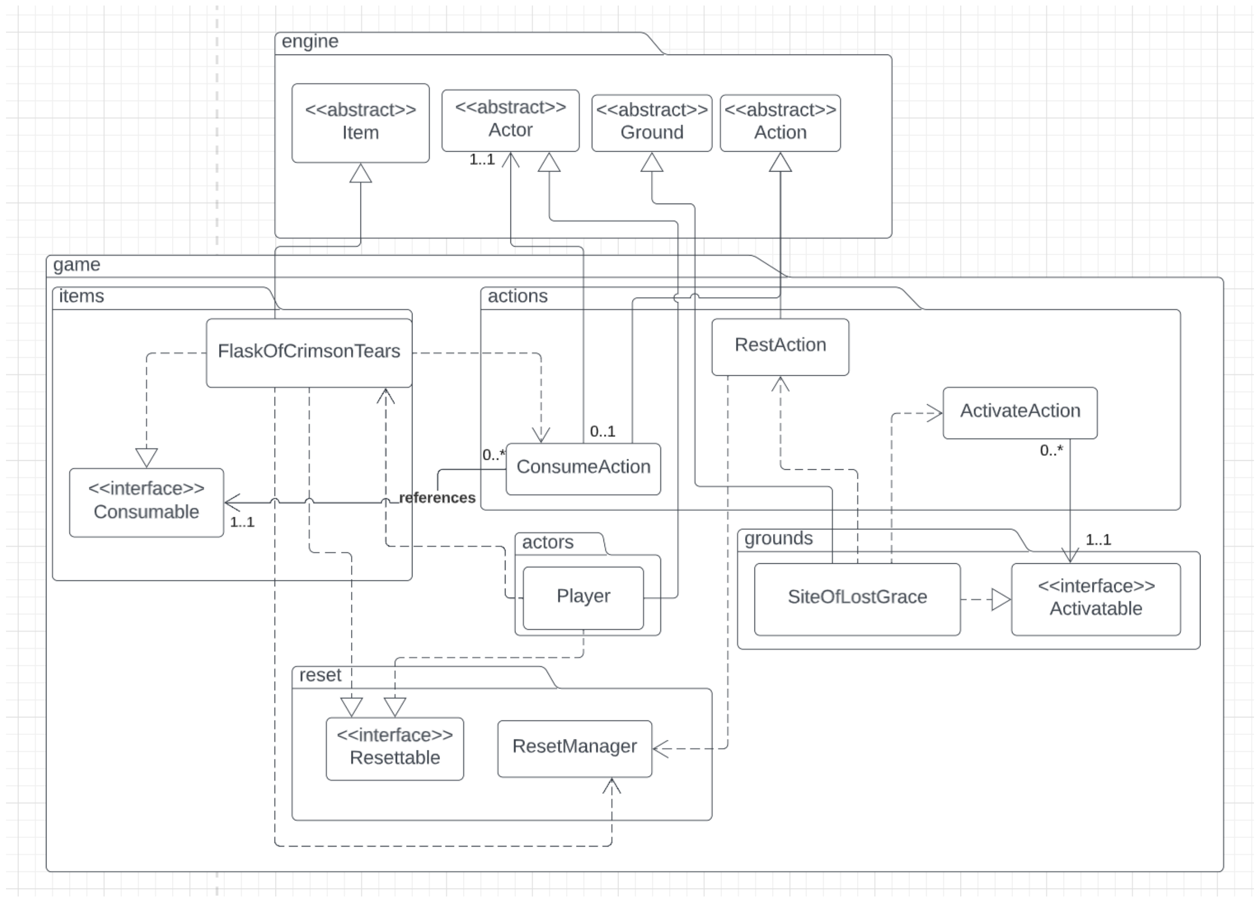
This design is extensible because if in the future the merchant has a new item to sell it can just create it and pass it thru the TradeManager.

Other design implantation choices:

- Using a buyable and sellable interface
    - Pros : In this way we remove extra dependencies for MerchantKale on any other weapon that we want the merchant to buy in the future
    - Cons: we have to do type checking and casting when adding the item to the players inventory
- Create weapons in Merchant
    - Pros : significantly reduce the number of dependencies

- Cons: it hinders extensibility because each merchant has their own list of weapons to sell

REQ3AB (Updated)



The figure in REQ3AB shows the following new classes and interfaces.

- Flask of Crimson Tears
- Consume Action
- Site of Lost Grace.

New additions as of Assignment 3

- Activatable
- ActivateAction

REQ3A

Since the Flask of Crimson Tears needs to be able to provide the player with the option to heal, it will be responsible for handing the engine a Consume Action. The way that Consume action interacts with the game engine relies on polymorphism and hence is a good example of following the Liskov Substitution Principle.

Flask Of Crimson also extends the Item abstraction which is another instance of Polymorphism.

A superclass for the flask could have been devised, however it would only serve to increase the amount of repetition since healing items are likely to be so different in implementation that a superclass doesn't exist.

REQ3B + Assignment 3 New Reqs

The Site Of Lost Grace, as a new ground tile will be extending the existing ground abstraction. Since the specific type of ground is likely to be unique, we chose not to further abstract the class as it would not assist making things DRY.

The site of lost grace will be responsible for advertising to the actor that the game can be reset, hence it will be dependent on the rest action.

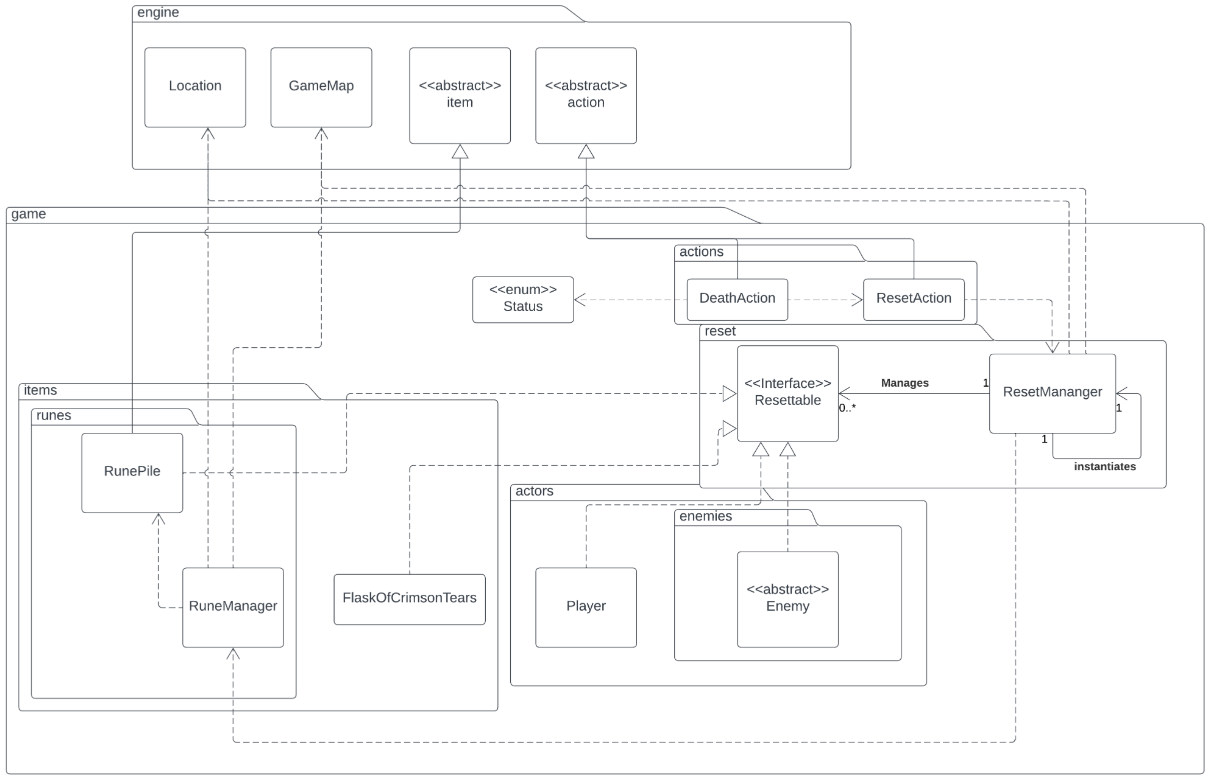
For similar reasons to the flask, the site of lost grace is very unlikely to have a similar class in future so further abstraction didn't make sense.

As of the **new Assignment 3 requirements**, the Site of Lost Grace is now required to be activated before use. To facilitate this, we have added an Activatable interface and a corresponding ActivateAction. This functions very similarly to the ConsumeAction in that the Site Of Lost Grace will be responsible for its own activation logic and will rely on Polymorphism and Action to interact with the engine.

The Activateable suite of classes can later be used to create other activatable ground tiles, actors or items.

We chose not to add de-activation functionality to the Activatable Class since doing so would violate the Interface segregation principle since SOLG isn't required to be able to de-activate.

REQ3CD



REQ3C:

There can only be one instance of ResetManager at a time (this is why it instantiates itself). It does this by storing the first instance it creates and returns it whenever an attempt to instantiate it again occurs. This makes sure that there is only one entity capable of resetting the game and no accidents can happen such as two ResetManagers existing where triggering one will only reset some objects

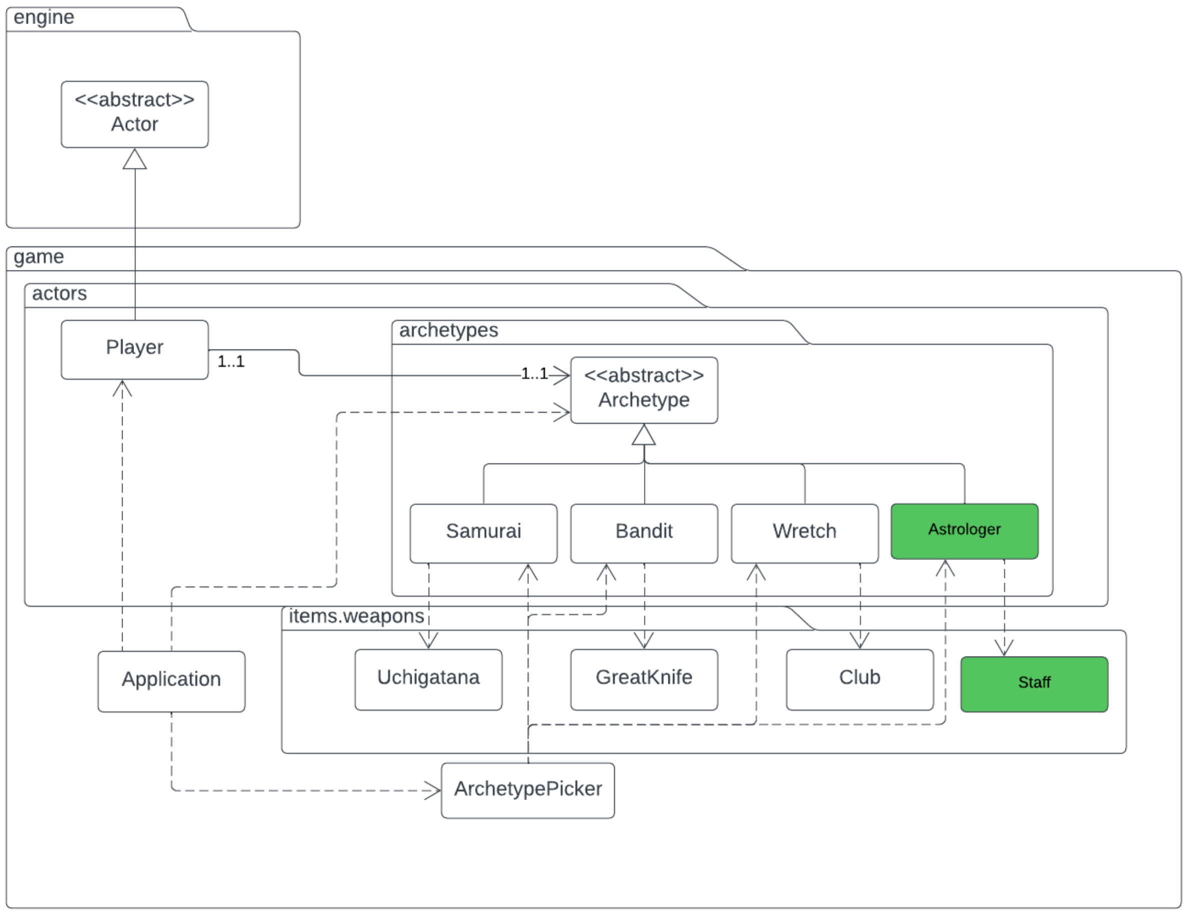
Through the use of the Resettable interface it gathers all instances it needs to reset into an array. This allows the reset manager to run the reset method without having to know anything about the classes it's resetting to avoid a dependency on all these classes. Conforming to SRP as the ResetManager only does one thing, Reset.

The ResetType enum is used to differentiate between a “soft” or “hard” reset (lost grace and death reset respectively) this allows us to avoid resetting somethings that shouldn't be reset.

Finally the ResetAction class is responsible for triggering the reset. Giving sole responsibility for the action to a single class instead of spreading it across DeathAction and SiteOfLostGrace.

REQ3D:
This requirement is simple, upon the player's death a RunePile is generated.
The RunePile Inherits the Item class to allow it to be picked up and interacted with.
The RunePile also implements the resettable class but it should only be reset when a hard reset is called. As it is only reset upon the death of a character and not a grace reset.

REQ4AB



The figure in REQ4AB shows the following new objects:

- Astrologer
- Staff

In this implementation Player is injected with an archetype at the beginning of the game. This will allow the player to determine the starting stats and items and should allow us to add other abilities to the sub-classes later.

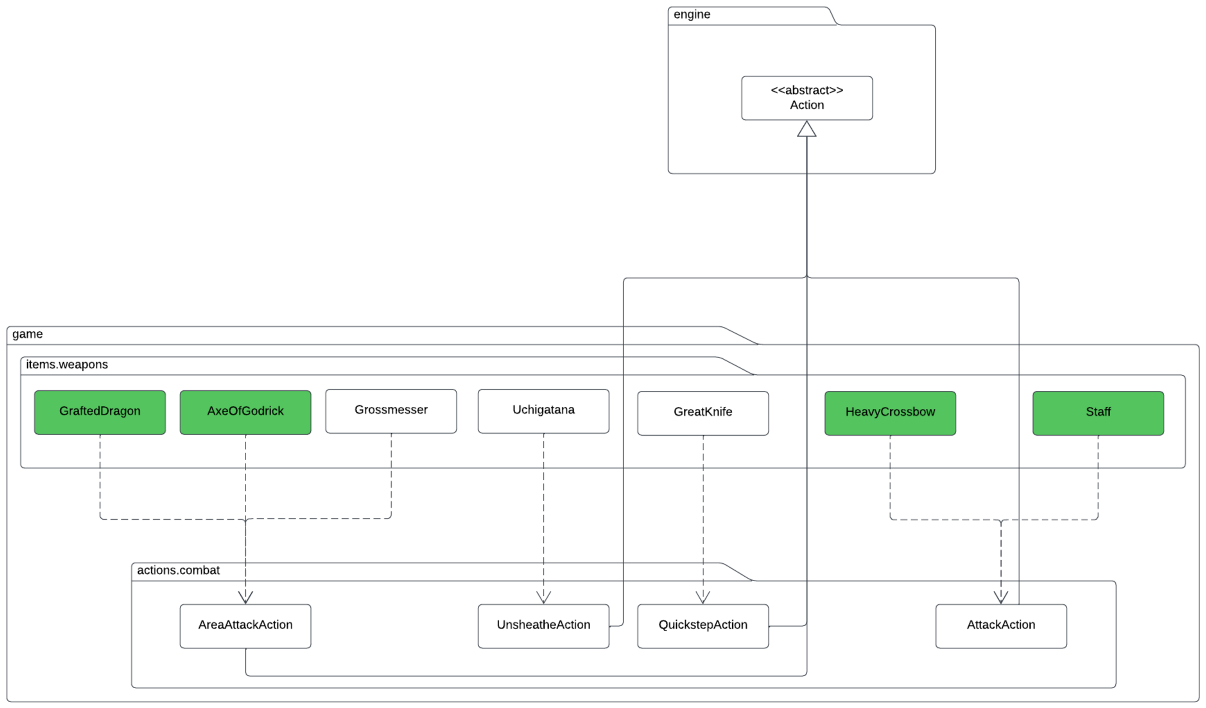
The generalizations also allow for Polymorphism of the archetype class irrespective of their concrete implementation which enables us to follow the open-closed principle as we do not want to have to changemuch/any code when we implement a new archetype in future.

Creating 4 new classes is also a good show of the Single Responsibility Principle as we are expecting each of the new subclasses to focus on the specific implementation of that archetype and leaving the job of defining actions that all players should have to the Player class.

These classes will be dependent on their respective weapons as they need to assign themselves one.

The Archetype picker will help take some of the strain off the application class by handling class selection. This will reduce the number of concrete dependencies that application has, and is an example of the Dependency Inversion Principle as it will now only have to rely on the Archetype class.

REQ4C



This figure in REQ4C shows the following new classes:

- GraftedDragon
- AxeOfGodrick
- HeavyCrossBow
- Staff

REQ4C

Each weapon creates its own unique action itself. Due to how the engine is built and to follow OOP principles it is better to attach each action to its relevant weapon. By doing that instead of creating them in the player, we reduce extra type checking to see what weapon the player is using hence reducing coupling between the classes. (ReD)

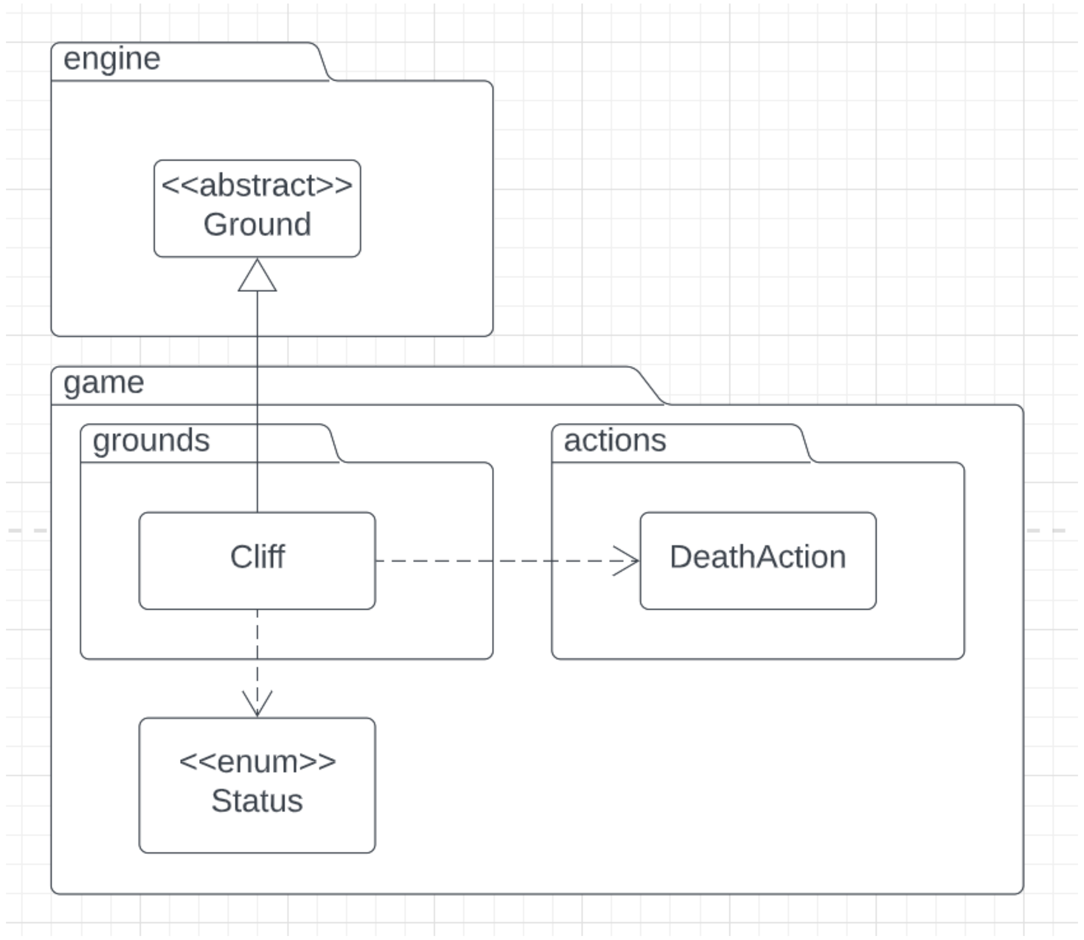
All Actions extend tha abstract class ‘Action’. Because all Actions need to be executable, have a hotkey, etc and only differ in the specificity of the action it makes sense to use the ‘Action’ class as a parent class to avoid writing the same code multiple times. (DRY)

Other design implantation choices:

- Actions are written from scratch:
    - Pros : each Action can be build so that it behaves exactly as intended with no extra or unused methods
    - Cons : writing a lot of repetitive code
- Actions are initialised in application and added to Weapons:
    - Pros : less dependency between Action and weapon
    - Cons : application would become a godclass and there would be lot of coupling between application, weapon and Action classes

Assignment 3 New Requirements

Cliffs



The above diagram shows and this section will discuss the following new classes:

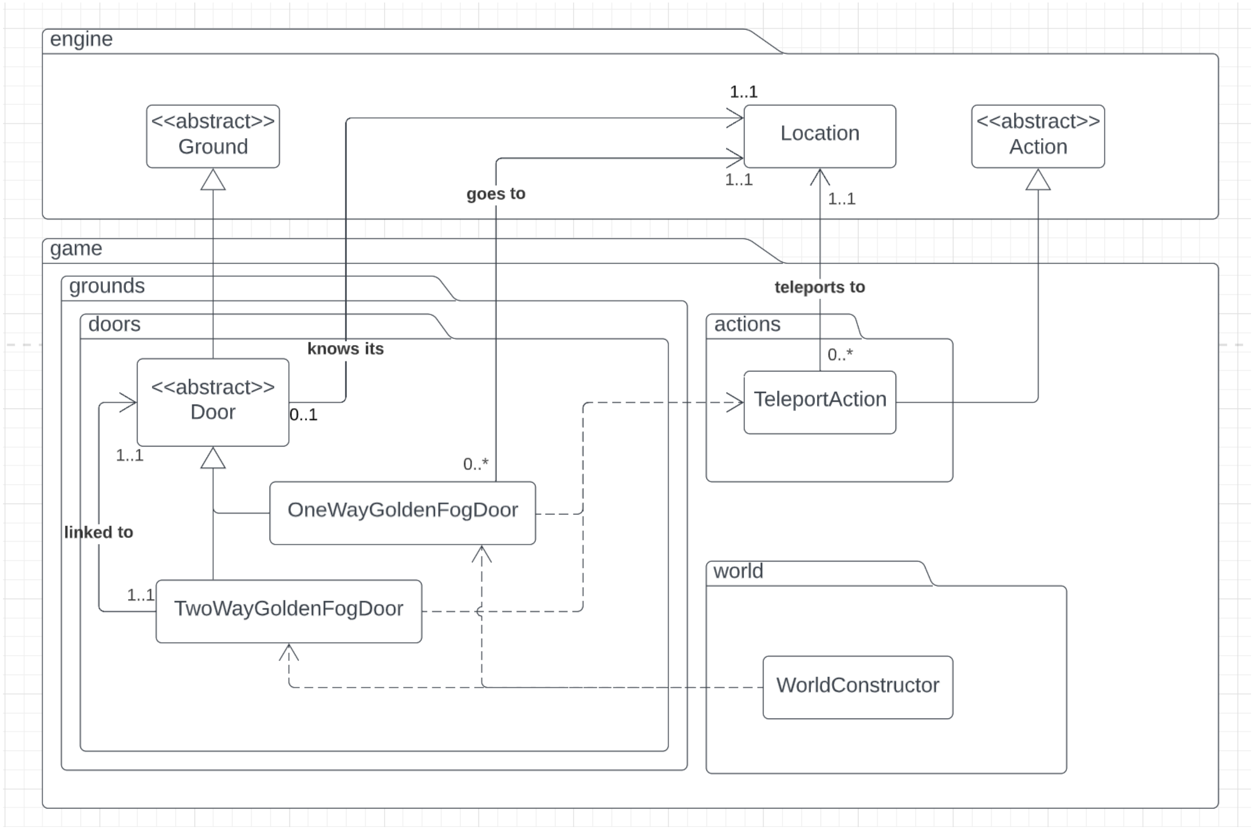
- Cliff

Cliff will be an extension of Ground as it will be placed on the game map as a normal ground tile would. For that reason, we need to ensure that the new Cliff class upholds the Liskov Substitution Principle.

We have chosen not to further abstract the Cliff since it is unlikely that there will be hazardous tiles in future that closely mimic Cliff's standard behaviour and hence would provide little to no DRY benefit. We have also chosen not to include a hazardous interface for the similar reason.

Cliff will be dependent on Status and DeathAction to allow it to perform its functionality. An alternative could have been to somehow integrate an interface for actors that are susceptible to hazards such as the cliff. But since there is no inherent benefit to having interface methods within the actor, and the fact that it would likely introduce a lot of type-checking, we decided against it.

Doors



The above diagram shows and this section will discuss the following new classes:

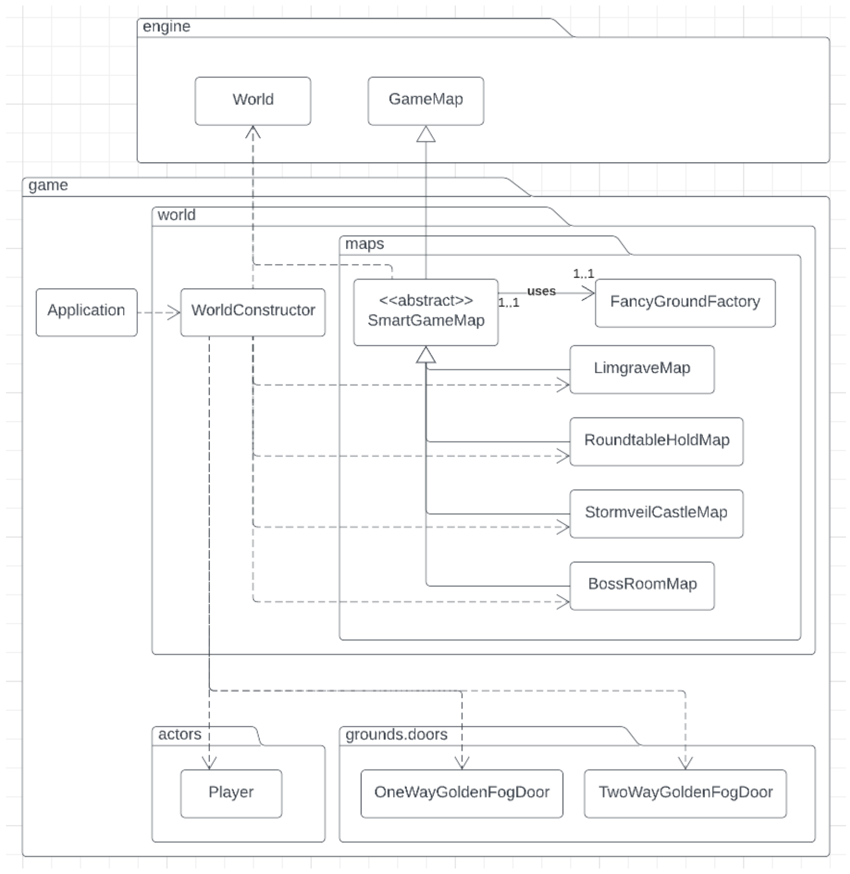
- Door
- OneWayGoldenFogDoor
- TwoWayGoldenFogDoor
- TeleportAction

Teleportation between two Locations on any GameMap will now be handled by the TeleportAction which is an extension of the Action class. By separating this functionality from the door classes, we can re-use this code elsewhere for anything else that may require teleportation later on. For these reasons, Teleport action is a good show of the Single Responsibility Principle and the Open-Closed Principle.

As the requirements indicated that there were doors that are linked and doors that only take you one way down a path, we have chosen to split the doors into 2 concrete classes. These doors will take the form of instances of Ground.

If for whatever reason, we needed to reference both door types in the future for the purpose of creating some sort of moving maze or if we wanted to link a 2-way door to a 1-way to create a puzzle, then we'd need to be able to have a reference for both door types. For this reason, we have chosen to create an abstract class "Door" to enable this kind of polymorphism.

New Game Maps



The above diagram shows and this section will discuss the following new classes:

- WorldConstructor
- SmartGameMap
    - LimgraveMap
    - RoundtableHoldMap
    - StormveilCastleMap
    - BossRoomMap

Since previously Map creation was handled by the Application class, continuing to expand upon it would begin to violate the Single Responsibility Principle and will generally lead to it being a Bloated/God class. For this reason, we have delegated this Responsibility to a new class named “WorldConstructor” which will be responsible for building the set of GameMaps and then linking them together. It will also add the player to the appropriate GameMap at the start of the game.

To further assist the WorldConstructor with its role, we have also delegated the responsibility of constructing elements that do not involve several GameMaps to a new “SmartGameMap” class.

Implementations of this smart map will have a structured way for them to define their own starting conditions so that WorldConstructor and Application do not have to be directly concerned with them. This further shows the application of the SRP and also is a better show of the OCP since you can now create and modify new game maps with little to no other modifications required for the higher-up classes.

Please note that the specific dependencies for each concrete SmartMap have been omitted for clarity.

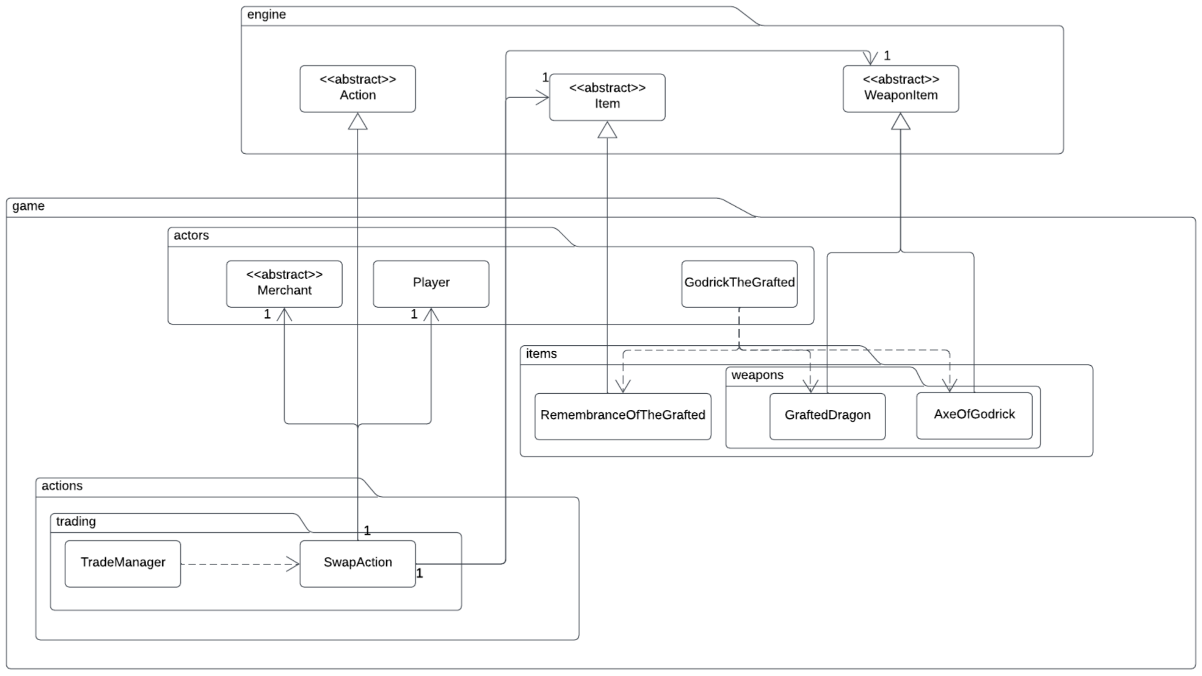
New Spawnable Grounds

Please see the updated REQ1A

Site Of Lost Grace Changes

Please see the updated REQ3AB

REQ3.A,B.2 Godrick the Grafted

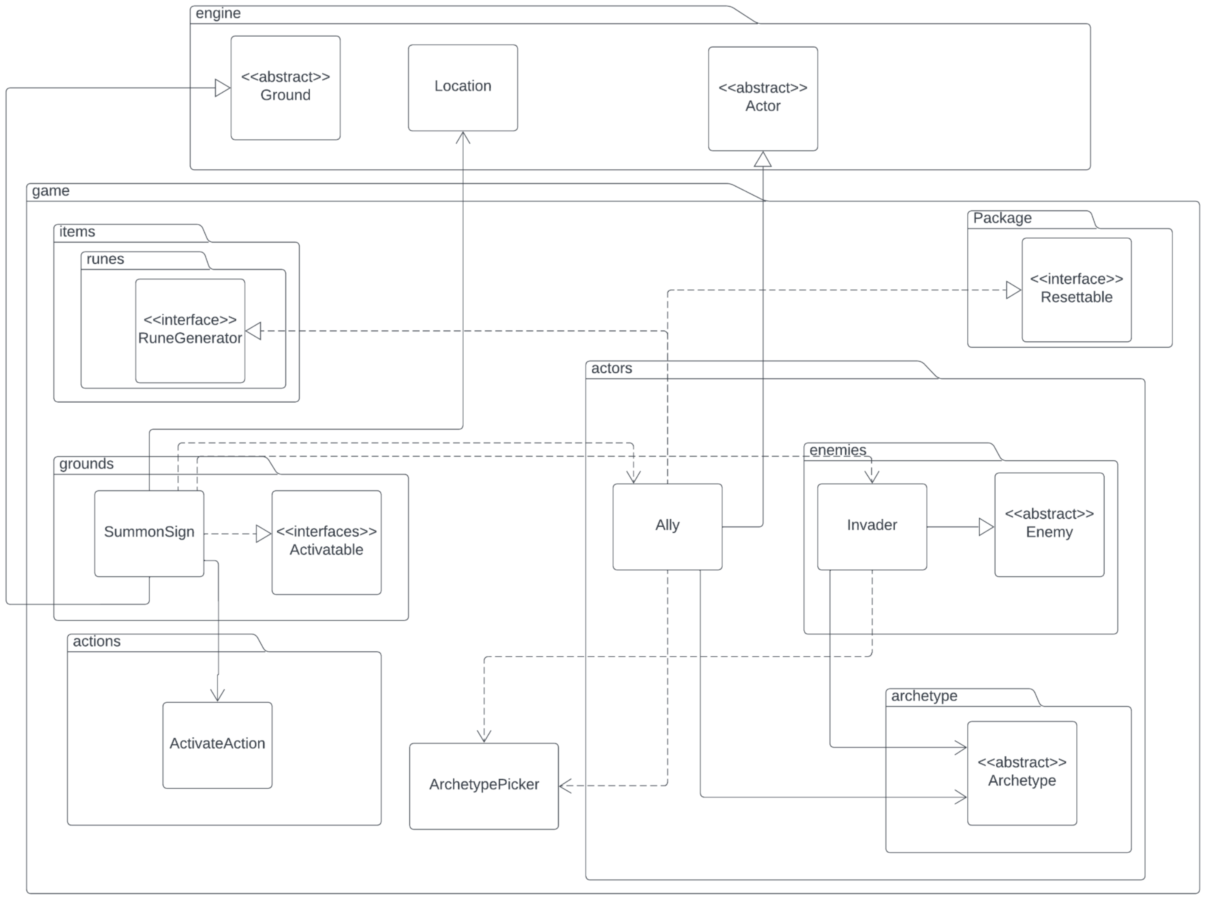


Enemies Create their own weapons and items that they use or spawn.In this way we alleviate the responsibility of creating them from the Application, so that Application would not have additional things to handle(SRP)

The TradeManager class follows the Single Responsibility Principle (SRP) by handling trade-related operations and ensuring a single instance exists at any given time

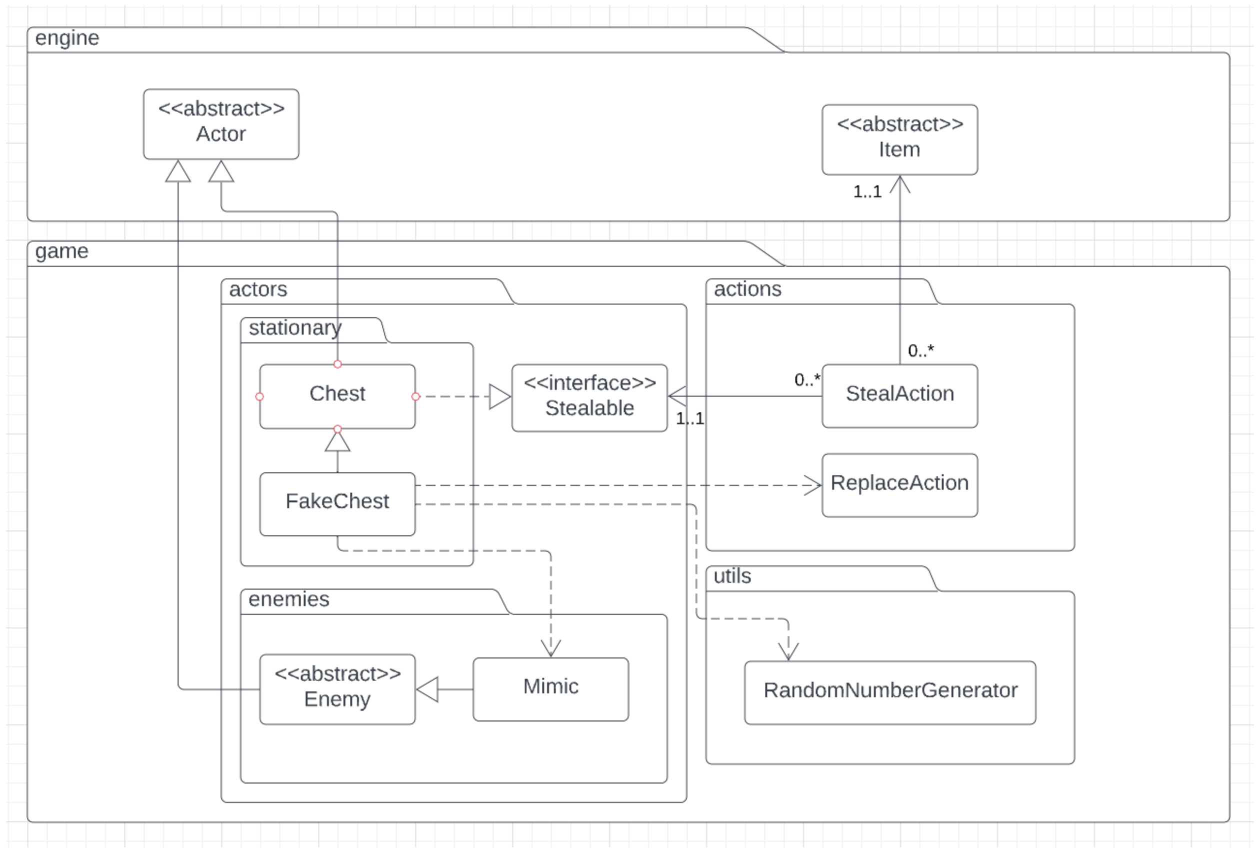
All Actions extend tha abstract class ‘Action’. Because all Actions need to be executable, have a hotkey, etc and only differ in the specificity of the action it makes sense to use the ‘Action’ class as a parent class to avoid writing the same code multiple times. (DRY)

Ally, Invader and its SummonSign



- Ally and Invader should rely on dependency injection using archetype picker for getting their loadouts
- Ally should not inherit the Enemy class because that would violate LSP and maybe ISP
- For the same reason as above SummonSign should not inherit spawnableground (it should spawn through the activatable interface
- Ally and Invader store their archetypes as an attribute to leave room for future extensibility (OCP)

Creative Requirement: Mimic and Chest



The above diagram shows and this section will discuss the following new classes:

- Chest
- FakeChest
- Stealable
- StealAction
- Mimic

Feature Description

Now, chests can be scattered throughout The Lands Between, from small caves to treacherous dungeons. Within these chests, the Tarnished can find items, potentially even permanent buffs and key items (Not included). However, the Tarnished must beware, for some of these chests are not real.

Upon attempting to take an Item from a fake chest, there is a chance that the chest spontaneously turns into an angry, sneaky and vicious Mimic.

(Dungeons & Dragons, 2014)

Mimics are very scary creatures as they have a 10% chance of dealing 10000 damage to the player and have 280 hitpoints. Upon death, Mimics drop the item that the player was attempting to grab from the Chest as well as exactly 1337 runes.

It is worth noting that the Mimic will not make use of a Weapon that it denies the Tarnished from collecting. Mimics do not wander the map as they need to act inconspicuously but will follow anyone that gets too close to them and will not despawn naturally. The Mimic's internal contents will be lost on a game reset/rest.

Rationale

Since Chests conventionally sit on top of Ground tiles and because the Actor class already has an Inventory implementation that we can use, we have decided to have Chest be a type of actor. This in turn makes our code follow the DRY principle while continuing to uphold the LSP.

Should we want other types of Trick Chests, we can simply extend the Chest Implementation.

FakeChest will be an extension of Chest that serves to attempt to spawn a Mimic when interacted with. Otherwise, this class will maintain typical Chest functionality. Since we have already implemented a ReplaceAction for swapping 2 actors out, we will be re-using this to facilitate the swap.

As Mimic is a hostile creature, it will be extending from the Enemy class. Since we have preexisting code for standard enemy behaviour, we are upholding the DRY principle. Note that, it will not become a part of a Family as it is not friendly with anything. This doesn't break the LSP as Enemies are not part of a family inherently.

The Stealable interface will describe Actors that can have Items stolen from their inventory. Since this will only be applied to actors that are able to fulfil this functionality, this is a good example of the ISP and LSP at work. To facilitate the stealing of Items from a Stealable's inventory, StealAction will be offered as an Action to the stealing party.

Unlike a normal PickupAction, StealAction will notify the Stealable that it is being stolen from, allowing that Stealable to determine the outcome. This ensures that the functionality employs the use of the SRP since the Stealable should describe its own implementation.

Potential Alternative Design Choices:

- Abstract StealableActor class/adding functionality to Actor (If we could)
    - Poor extensibility.
    - Violates LSP.
- 2 StealActions for each type of Item (Item, WeaponItem)

    - Increases repetition/redundant code.
    - StealAction has no need to reference Weapons directly and allowing it to do so would likely violate SRP.
- Chest as a ground class
    - Would need to have its own inventory when we already have an inventory implementation (WET).
    - Grounds shouldn't spontaneously convert themselves into Actors. Seems like a design smell even if the implementation may work fine. Additionally one could argue that having the ground tile disappear would violate LSP.
- Abstract Chest
    - Generalization for the sake of generalizing.

References

Dungeons & Dragons. (2014). *Monster Manual (D&D Core Rulebook)*. Wizards of the Coast Publishing.