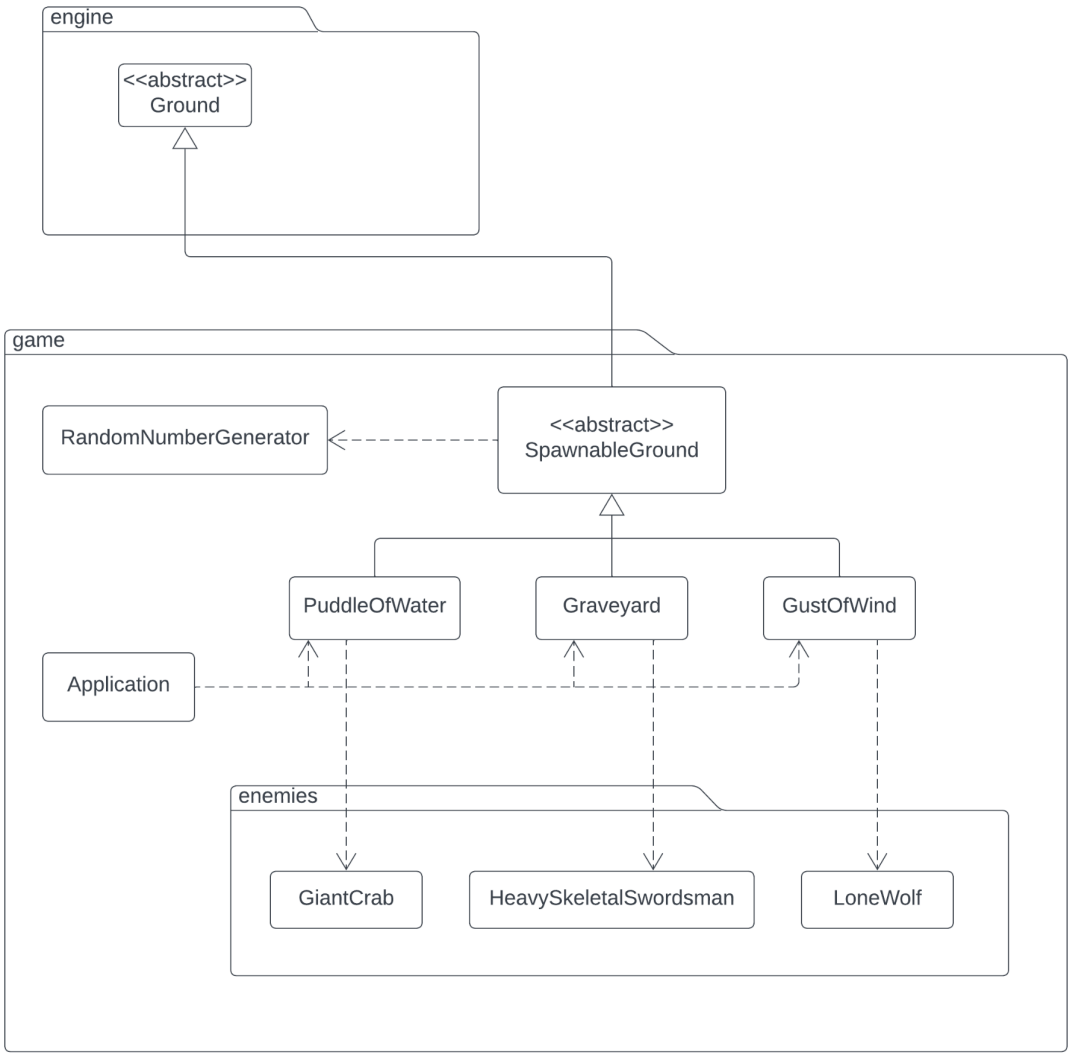
Design Rationale Assignment 1

William, Salar and Ibrahem

REQ1A



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)

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.

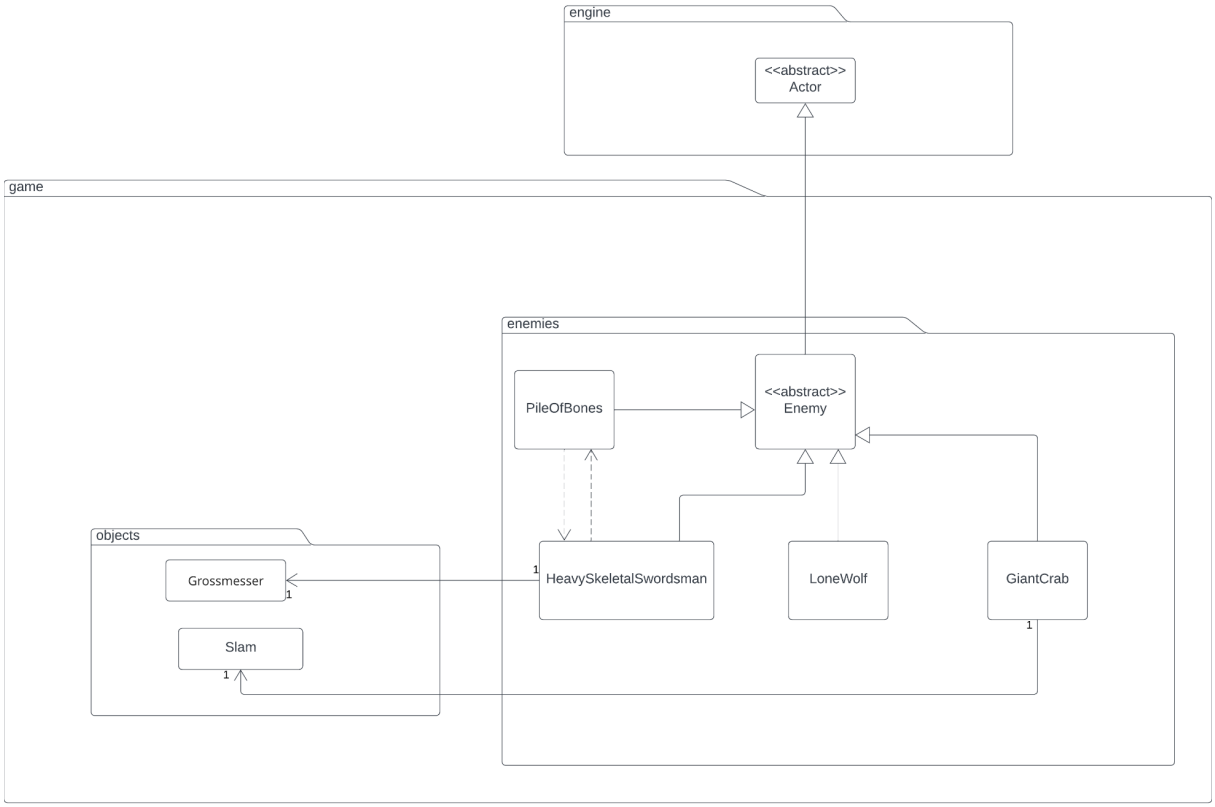
As per the design criteria, the new ground extensions will be responsible for spawning the respective enemy to the GameMap. Hence each new child of ground will be dependent on the respective enemy type.

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.

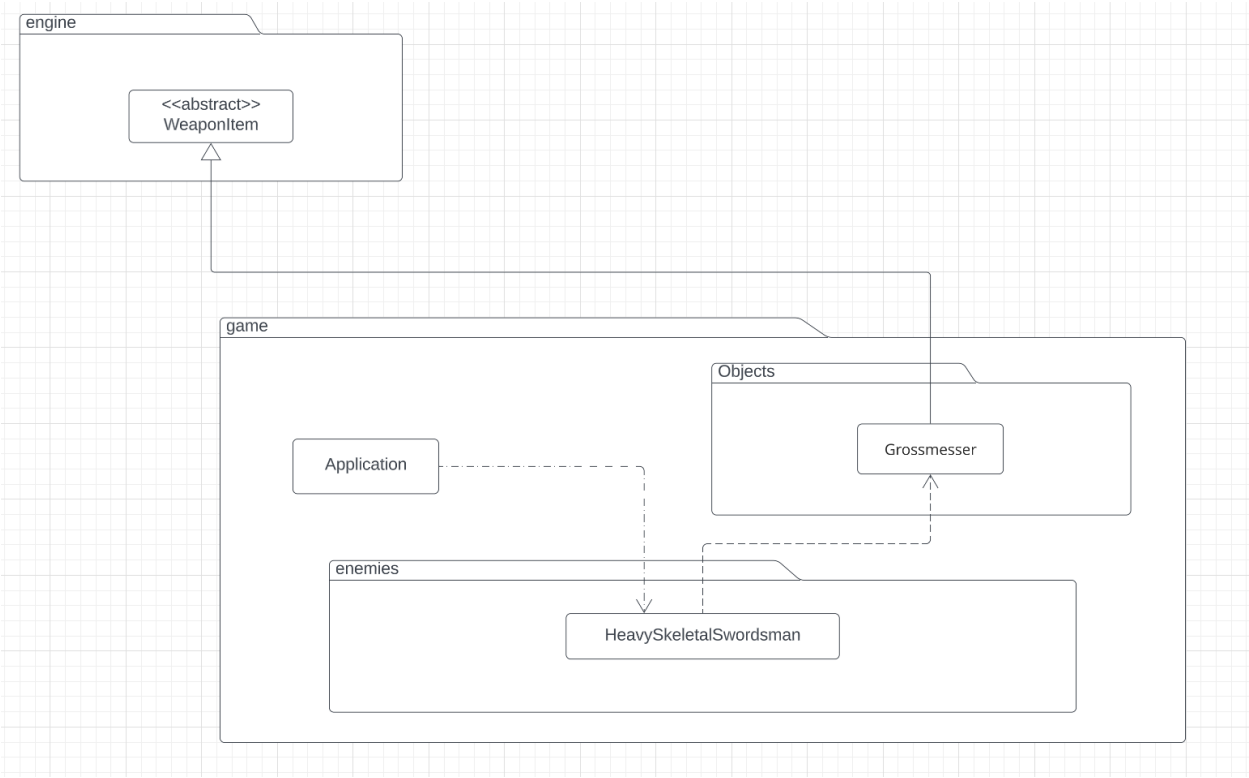
REQ1B



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:

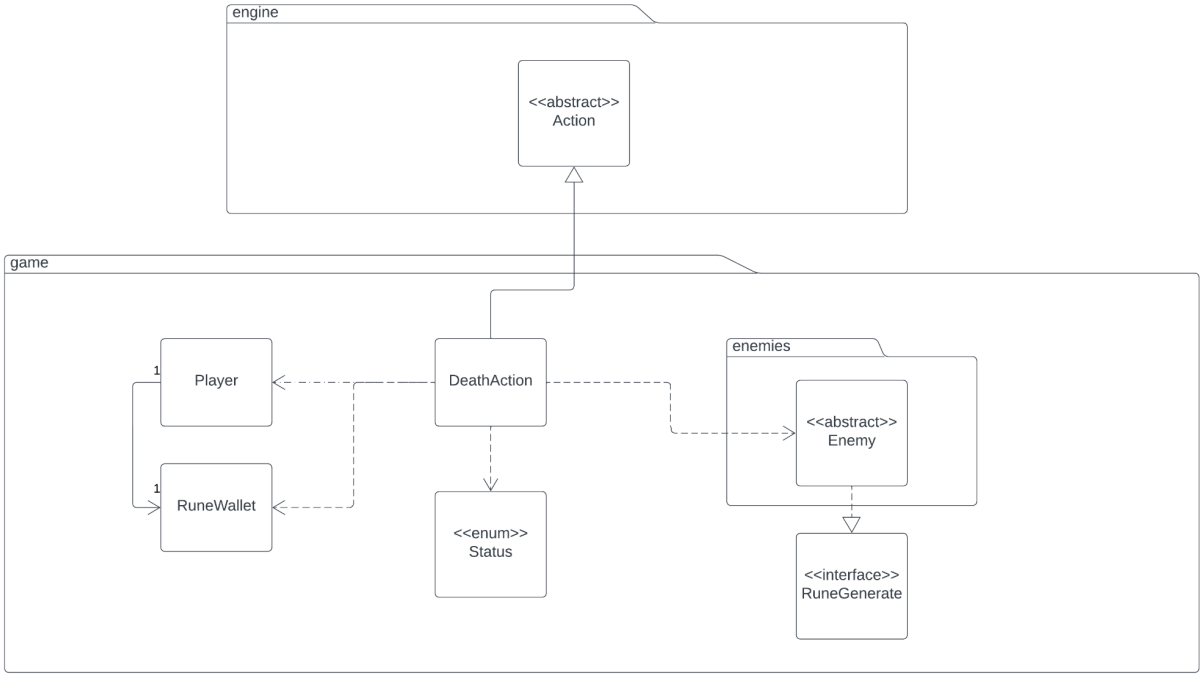
- Grossmesser
- Heavy Skeletal Swordsman

REQ1C

Grossmesser extends the WeaponItem abstract class since Grossmesser is a weapon and needs to have all the attributes and methods that a WeaponItem has. (DRY)

HeavySkeletalSwordsman(HSS) depends on Grossmesser and is not initialised by application. In this way we leave the responsibility of creating Grossmesser to HSS instead of the Application, so that Application would not have additional responsibilities (SRP)

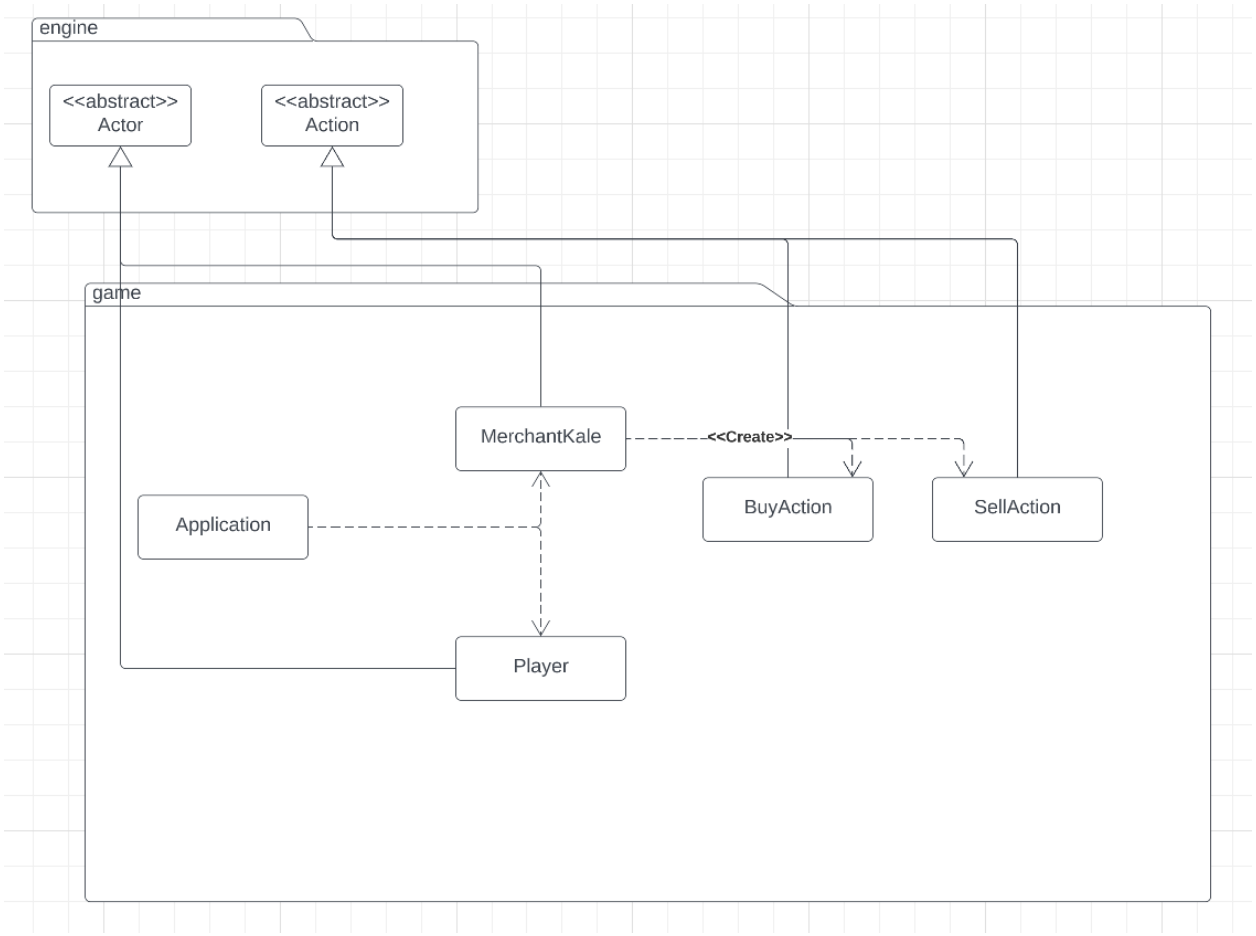
REQ2A



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:

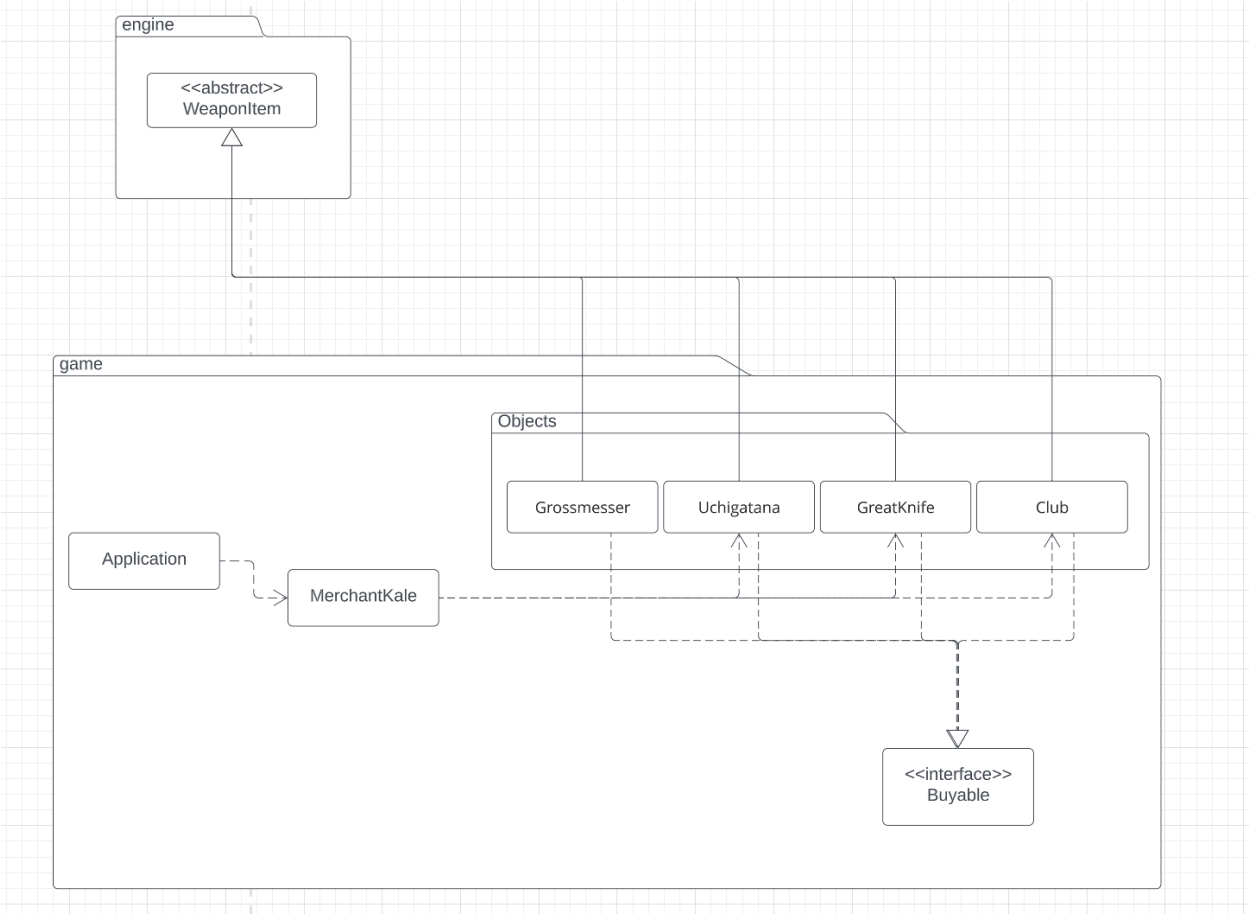
- MerchantKale
- BuyAction
- SellAction

MerchantKale extends the Actor class since it needs to use the necessary methods provided by ‘Actor’ to handle its inventory, Location, etc. (DRY)

MerchantKale <<create>> BuyAction and SellAction. Since in this game MerchantKale provides the player with actions it would be appropriate to attach the 2 actions to MerchantKale. By attaching the actions to MerchantKale instead of the Player we reduce the need to check who the Actor that the player is interacting with is and if the Actor is the appropriate Actor to start trading with, hence we reduce unnecessary dependencies (ReD)

Player doesnt use an Enum to check who is interacting with MerchantKale because it is assumed that MerchantKale is located on safe grounds that only the Playercan access.

REQ2C



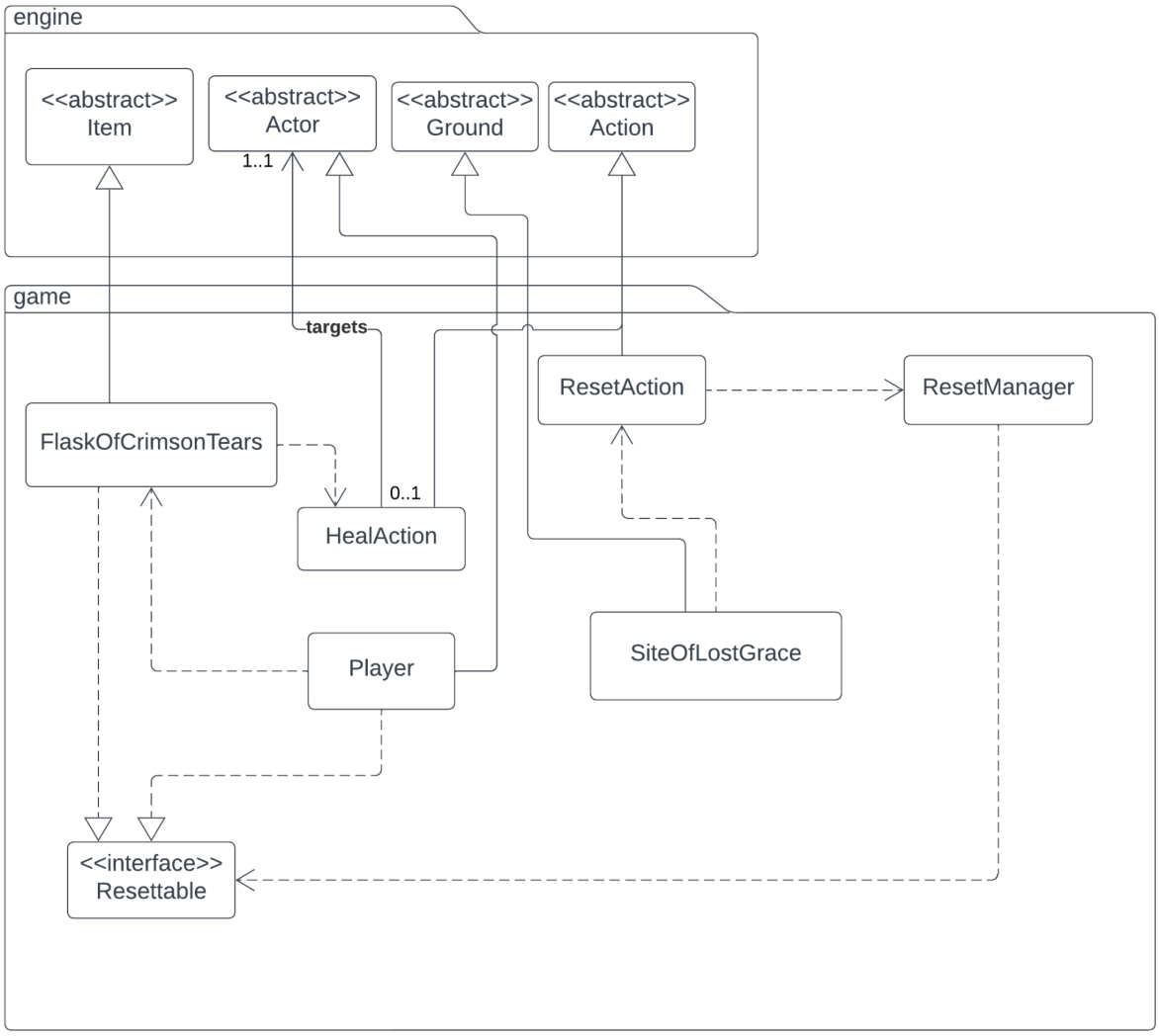
This figure in REQ2C shows the following new classes and interface:

- Uchigatana
- GreatKnife
- Club
- Buyable

MerchantKale creates and keeps an instance of Uchigatana, GreatKnife and Club in its inventory. Because we need to check the inventory and see what is available to buy, MerchantKale 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.

All weapons (included in the diagram) use the Buyable interface. In this way we remove extra dependencies for MerchantKale on any other weapon that we want the merchant to buy in the future. When adding a new weapon we do not need to modify MerchantKale to be able to buy them, instead we rely on the interface (Open-Close Principle).

REQ3AB



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

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

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 Heal Action. The way that Heal 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.

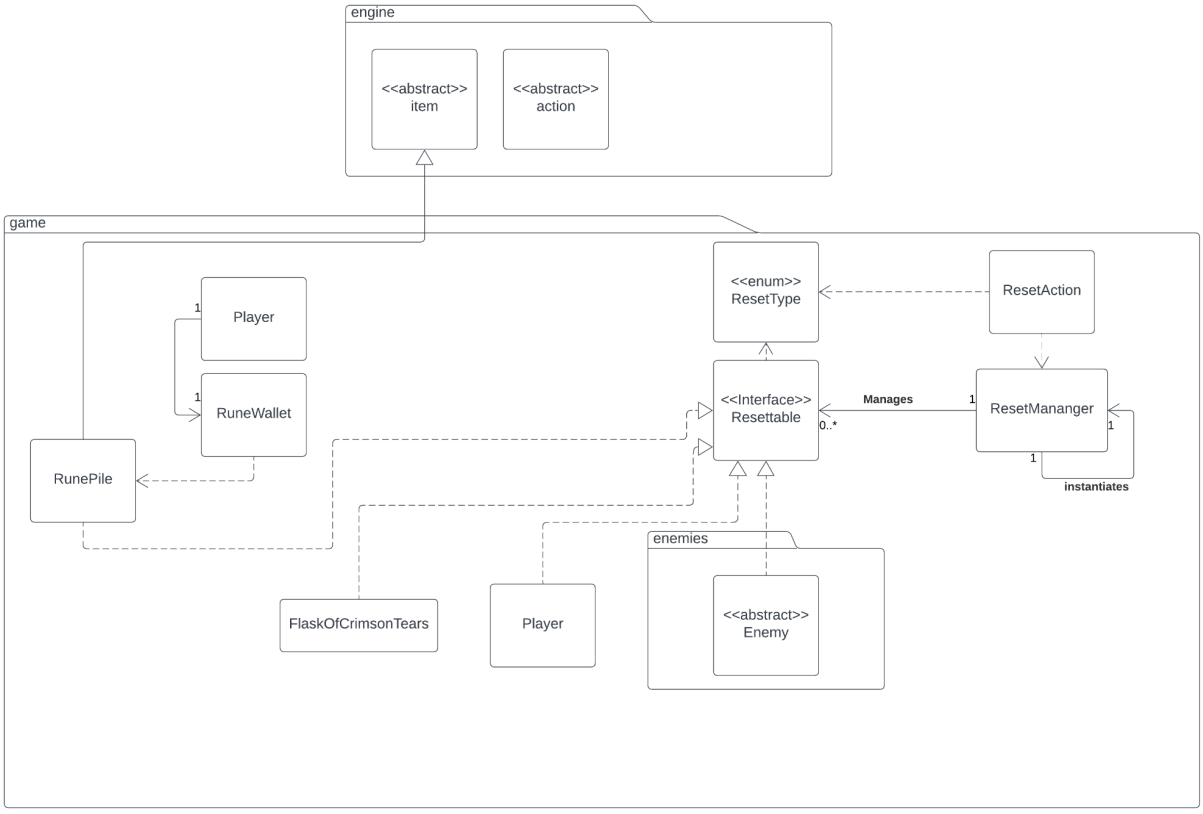
REQ3B

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 reset 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.

REQ3CD



REQ3C:

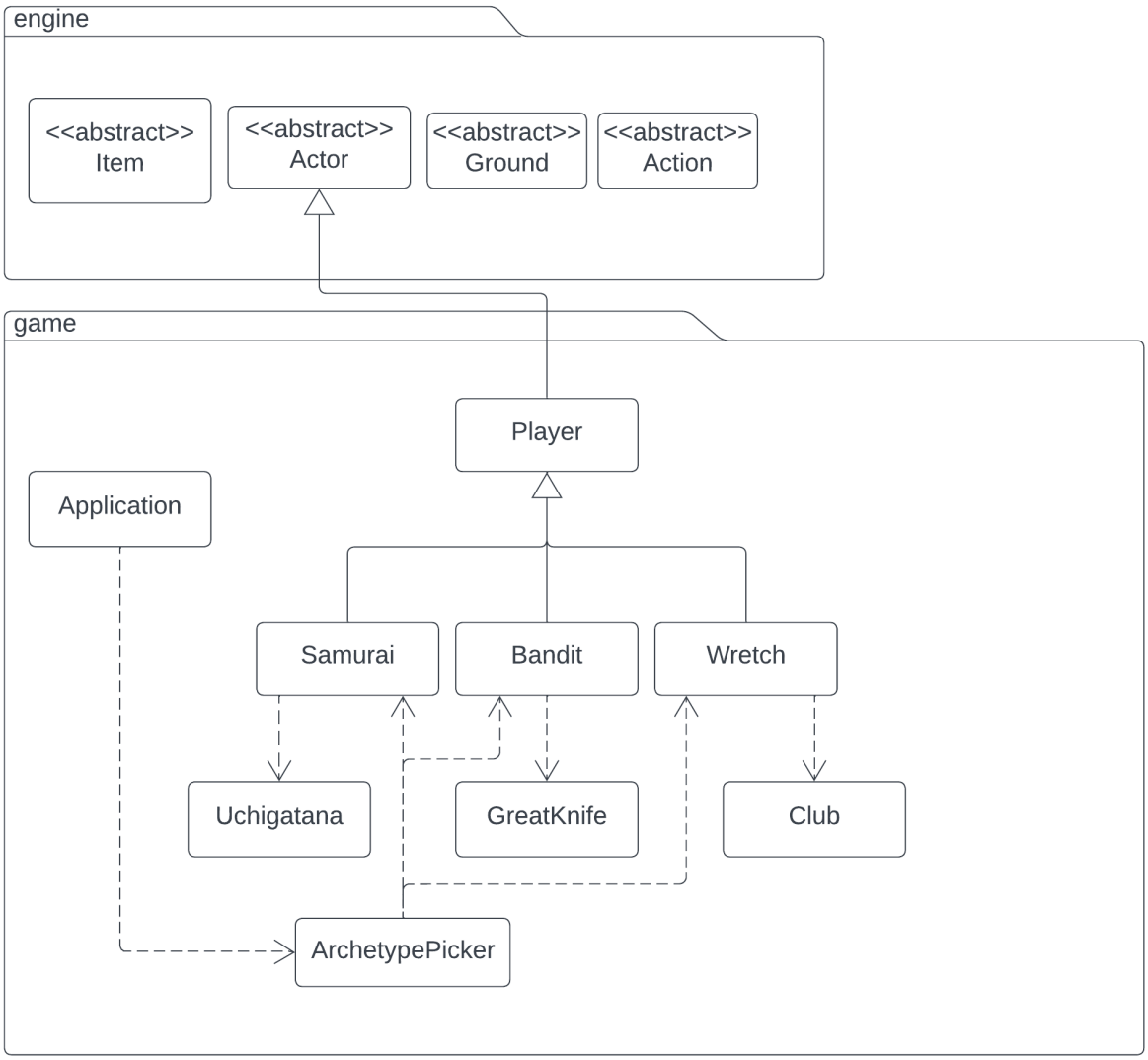
There can only be one instance of ResetManager at a time (this is why it instantiates itself). It does this by storing the first instance it creates and returns it whenever an attempt to instantiate it again occurs. This makes sure that there is only one entity capable of resetting the game and no accidents 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:

- Samurai Class
- Bandit Class
- Wretch Class
- ArchetypePicker Class

In this implementation Player is now extended by the 3 new classes. At the moment, there is limited functionality, but the extension of the existing player class will allow us to add much more functionality.

The 3 subclasses of Player should be able to do everything that an instance of Player does unless otherwise specified. This abstraction shows how this specific design follows the Liskov Substitution Principle.

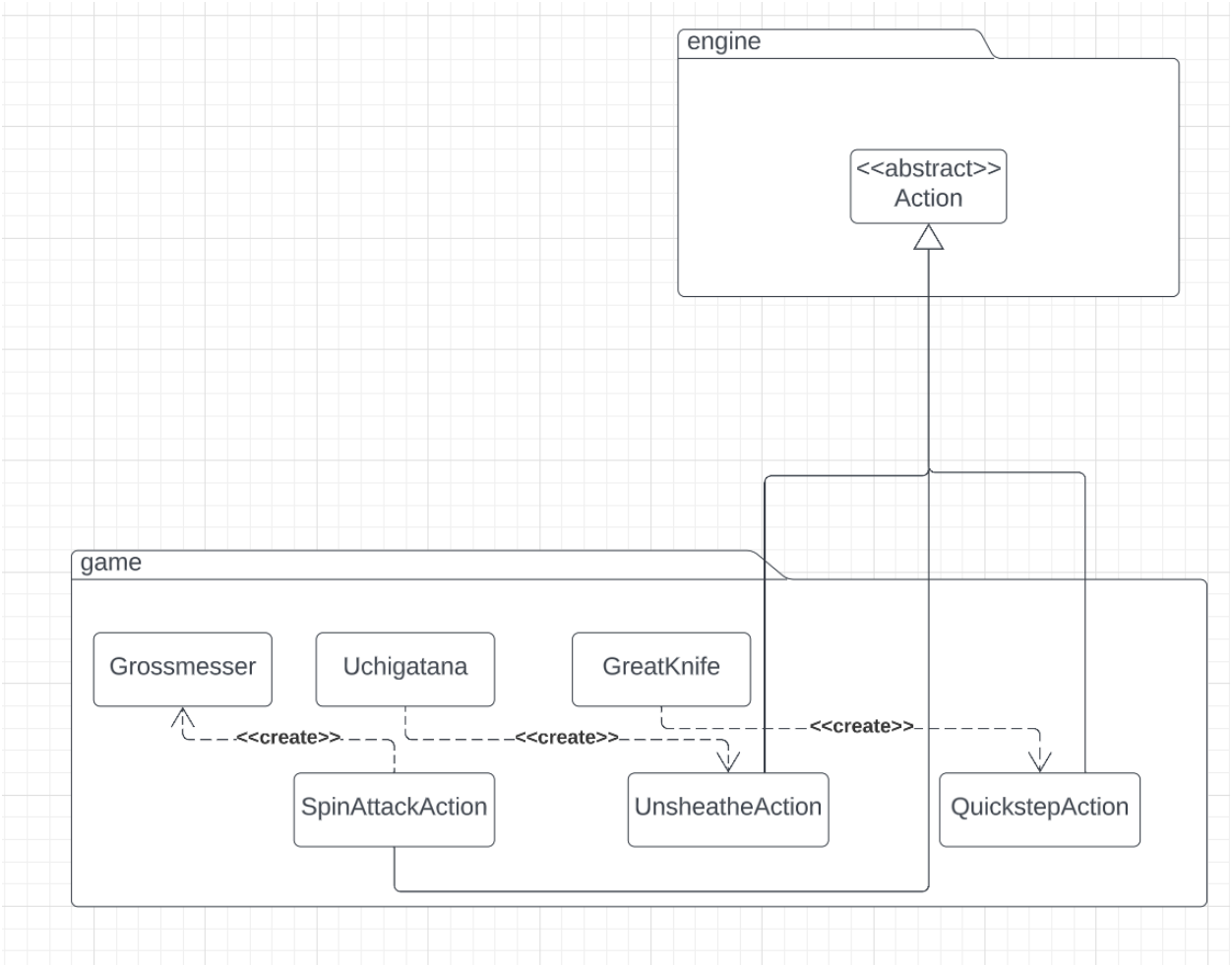
The generalizations also allow for Polymorphism of the player class irrespective of their class which enables us to follow the open-closed principle as we do not want to have to change code when we implement a new archetype in future.

Creating 3 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 player class.

REQ4C



This figure in REQ4C shows the following new classes:

- SpinAttacAction
- UnsheatheAction
- QuickstepAction

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)