Design rationale: why this approach is good? what's the alternative?

To help us understand how your system will work, you must also write a *design rationale* to explain your choices. You must demonstrate how your proposed system will work and *why* you chose to do it that way. Here is where you should write down concepts and theories you've learnt so far (e.g., DRY, coupling and cohesion, etc.). You may consider using the pros and cons of the design to justify your argument.

The design (which includes *all* the diagrams and text that you create) must clearly show the following:

- what classes will exist in your extended system
- what the roles and responsibilities of any new or significantly modified classes are
- how these classes relate to and interact with the existing system·
- how the (existing and new) classes will interact to deliver the required functionality

You are not required to create new documentation for components of the existing system that you *do not* plan to change.

Writing your Design Rationale

Here are some things to keep in mind when writing your design rationale:

- Briefly provide a summary of the projects, stating all your design goals (must be clear and specific).
- Explain the overall concept of your design
  - What is it
  - Who is the audience
- Once you have explained your goals, you can then go into detail, giving reasons for specific design decisions that you have made.

Dos

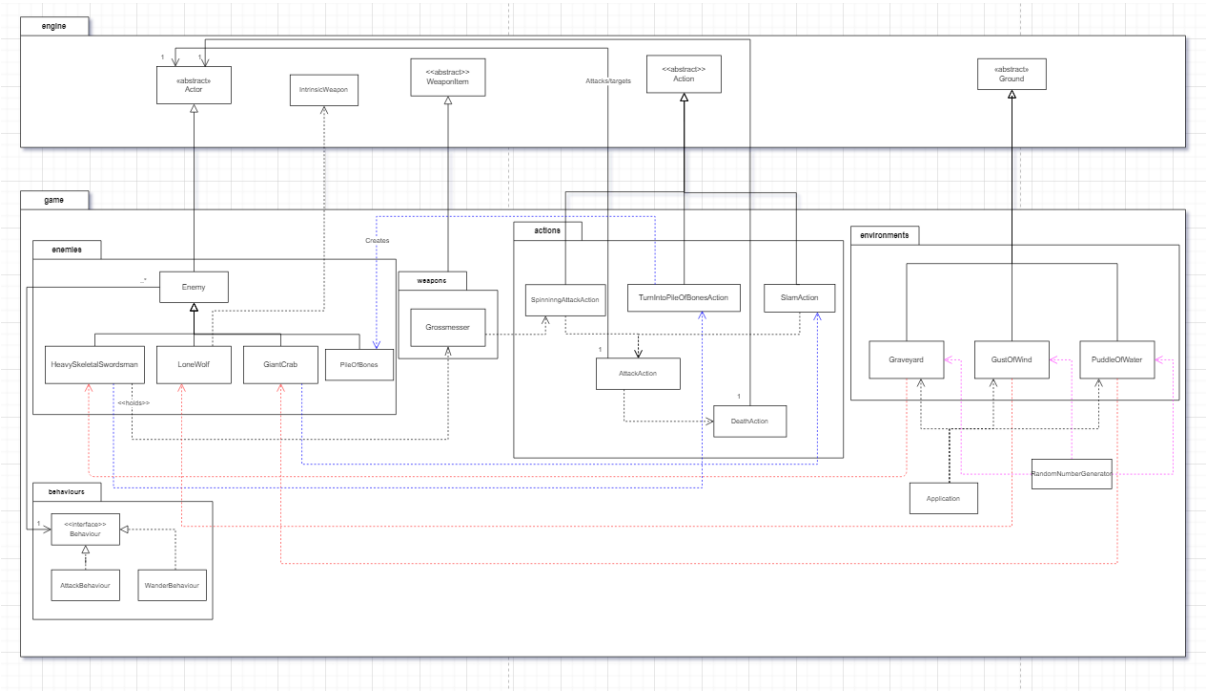
- Be clear and concise, you may use bullet points in this document.
- Describe how your design aligns with the goals that you stated previously.
- Comparing your decision with the alternatives, including both positive and negative aspects.
- Besides all the great aspects of the current design, also include some limitations and compromise. You can propose a solution to them or indicate what the tradeoffs are.

Don'ts

- Try to avoid passing judgment on your own work, either positive or negative. It's necessary to say **why** your solution satisfies the brief, but statements such as *"this solution is very great"* or *"this isn't my best work"* are not helpful.
- Describe the product prototypes without explaining the rationale and benefits of your decisions. Remember – it's a *rationale* (requiring the **reasons** or **logic** behind your decisions), not a description.
- Reasoning using your personal preference, for example, "*I make this interface because I think it will suit*".
- Blindly follows all design principles and states the benefits of them that you have extracted from the definition. Please be more specific about why you need it in your scenario.

—----------DELETE ABOVE THIS LINE BEFORE SUBMISSION—----------

Requirement 1



This diagram represents the system of environments and enemies that exist in the game.

Design Goals:

- Create the Environments, Graveyard, Gust of Wind and Puddle of Water that can spawn their respective enemies
- Create the Enemies (Heavy Skeletal Swordsman, Lone Wolf, Giant Crab) which can have: a unique ability, intrinsic weapon, or carry a weapon. These things may have different actions/capabilities
- Implement a way for enemies to be despawned when they are not following a player
- Create the Grossmesser weapon that can be dropped by the Heavy Skeletal Swordsman

Environments:

For the implementation of environments, we have extended Environments, Graveyard, Gust of Wind and Puddle of Water from the abstract Ground class to hold the common attributes and methods described in the class. These locations have a dependency relationship with RandomNumberGenerator which makes use of already existing methods so that we may implement a way for enemies to be spawned.

Enemies:

For our implementation of enemies, we have created an abstract class called ‘Enemy’ which extends the ‘Actor’ class originating from the engine. This extension is done to follow DRY principles, as we know that all enemies will share similar

attributes/capabilities such as attacking a player, following them, wandering about and despawning. All enemies extend these common behaviours and will be able to add their own unique abilities, such as the PileOfBones for the HeavySkeletalSwordsman.

We have added PileOfBones to extend from the Enemy class as it can be hit by the player or other enemies like any other Actor. With this we have also created the action ‘TurnIntoPileOfBonesAction’ that has dependency relationships with HeavySkeletalSwordsman (the swordsman is dependent on the ability) and PileOfBones (the action will create the bones).

For this current requirement, each enemy has a dependency relationship with their respective environments. This is due to the fact that each environment (Graveyard, Gust of Wind, Puddle of Water) passes their respective enemies that they spawn as dependencies into the tick() method.

Weapons and Attacks:

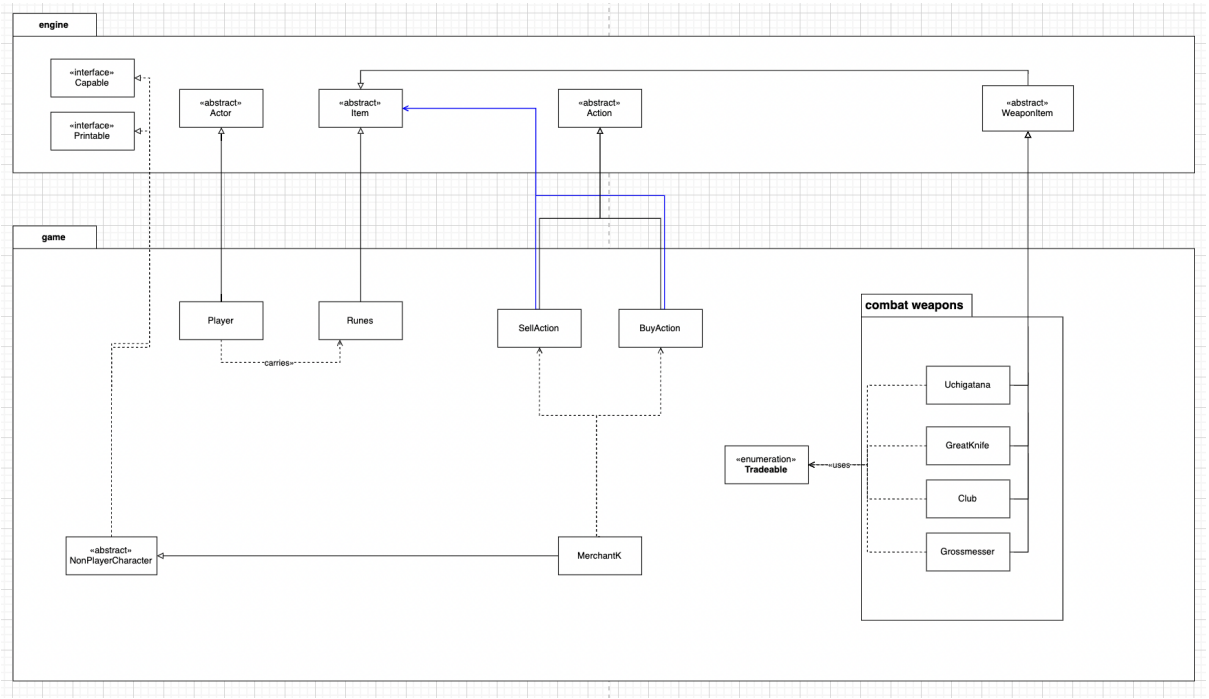
For our implementation, the Grossmesser extends the WeaponItem abstract class as it can be dropped and is a weapon. WeaponItem implements the Weapon interface which has a getSkill method, allowing weapons to have abilities.

To handle the area attacks in SpinningAttackAction and SlamAction, for every actor in the area that is hit, a new AttackAction will be executed. For this implementation, these attacks will use a getSurroundingActors() util method which returns a list of actors in the surrounding area to be targets of the area attack.

Potential Improvements:

- We could introduce an abstract class for the environments to extend from as in this requirement as they all ‘Spawn’ something. This will allow us to repeat less code as we know that the locations will all have a similar spawn method
- For future introduction of enemies that don’t despawn, we could add an interface for despawning, allowing us to easily create enemies that can and don’t despawn.
- A way to introduce and show the actor dying or reviving needs to be implemented. This could be via an interface which has a method that runs whenever an Actor dies. The same could be done with reviving, making an interface for PileOfBones which runs when it has not been hit for 3 turns (This is because later there will be a skeletal bandit which also has the same ability, not repeating code).

Requirement 2



This diagram represents the object oriented system for the trading and runes functionality in the game.

Design Goals:

- Implement a currency called ‘runes’ that can be used by the player as a currency
- Implement a system which the player can buy and sell from a merchant
- Allow enemies to drop a certain amount of runes

To avoid repetitions (DRY), a number of inheritance relationships exist in this system, namely:

- **MerchantK** extends the NonPlayerCharacter abstract class. Although it would be possible to inherit from the Actor abstract class as MerchantK shares similar methods and attributes to other actors in the game (eg. allowable actions, name, display character), they also don’t use many methods that are part of the Actor abstract class such as adding weapon to inventory, hurt, health points. Hence, creating a separate abstract class for Merchant K to inherit follows the interface segregation principle, as the NonPlayerCharacter abstract class would only implement attributes and methods from the Actor class that are relevant to NPCs such as MerchantK (ie. remove methods and attributes related to combat). We would also follow the liskov substitution principle as MerchantK would no longer be expected to fulfil combat-based expectations attached to the Actor class.

  Additionally, if we want to add more non-playable characters in the future, such as a blacksmith, it could inherit from the NonPlayerCharacter class and add any unique attributes/methods specific to the new character, making the system extensible and endorsing the open close principle.

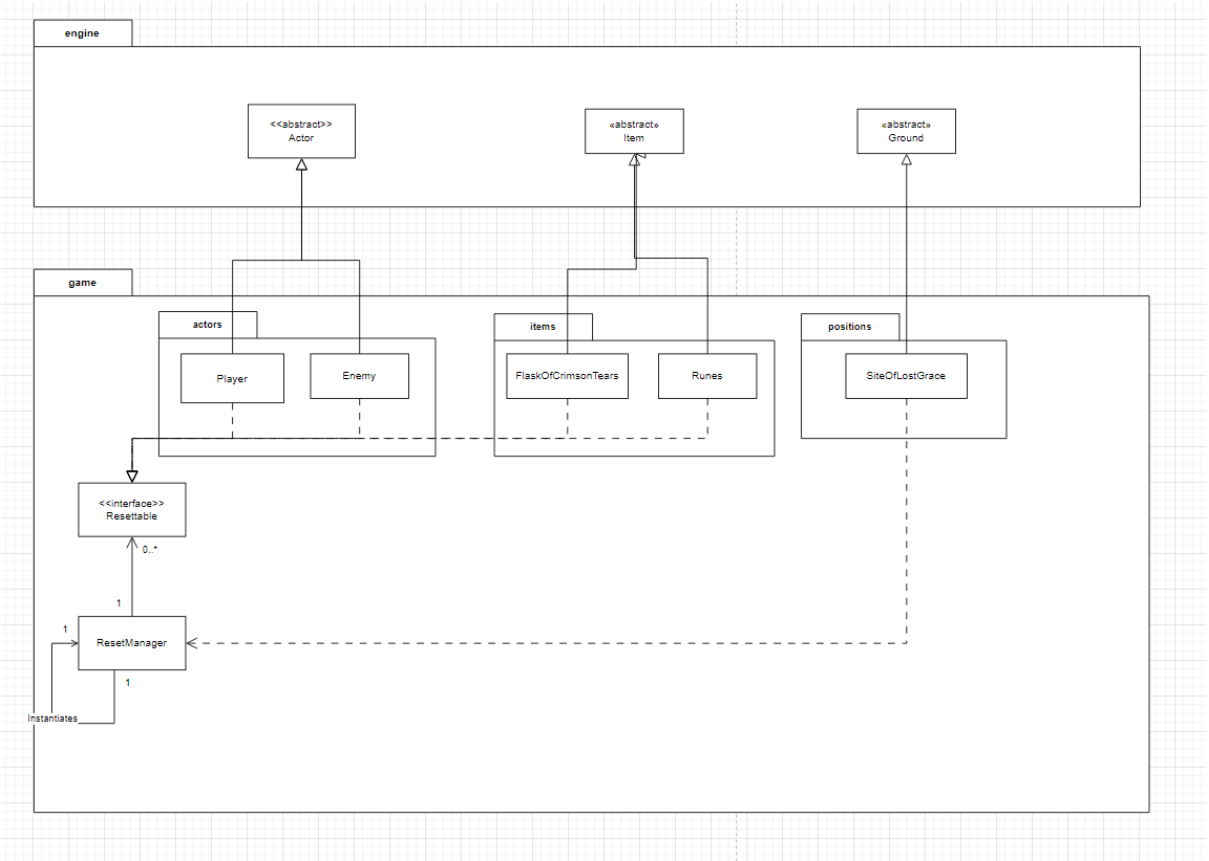
- **Runes** extends the abstract Items class as it shares capabilities similar to other items in the game such as pick up, drop. It would also have the ability to be added to an actor's inventory just like other items in the game.
- **BuyAction** and **SellAction** extends the Action abstract class as they share similar methods with other actions in the game (eg. execute). They also form associations with the Items class as each instance of BuyAction/SellAction will require a parameter of type Items in order to execute the trade.

  Alternatively, we could hold the player, merchant and items as attributes in the two actions, then implement a getter function in MerchantKale that will be able to access the items to buy/sell and their prices.
- **Uchigatana**, **GreatKnife**, **Club** and **Grossmesser** inherit from the WeaponItems abstract class as they all share methods and attributes included in the WeaponItems abstract class.

Weapons and Tradeable enum: Each tradeable weapon has a dependency on the Tradeable enumeration as this enumeration is added to the list of capabilities for each weapon. The tradeable enumeration indicates whether a weapon can be bought, sold, or both. This design supports the OCP as any items that are added in the future can use this enumeration to make the item tradeable without the need to modify existing code.

Alternatively, buyable and sellable could be made into its own abstract classes, and each weapon could inherit from either buy/sell abstract classes. However, this implementation would not work as some weapons can be both bought and sold, but classes can only extend from one abstract class. Alternatively another way achieve abstraction would be to use interfaces, for which a class can implement multiple if they need to, or just implement one. This follows the Interface Segregation Principle as weapons that can only be sold but not bought do not have to be exposed to methods related to buying. However, we have seen in that game that an actor inherits the capabilities of the items they pick up. Hence, if the player picks up an item that is sellable, they now have the capability to sell that item. We wanted to take advantage of this built in capability, and so we decided to choose enumeration over interface.

Requirement 3



Design Goals:

- Implement an item called ‘Flask of Crimson Tears’ that the player will always have, (cannot drop, always start with) that heals the player a certain amount of times
- Implement a game reset which despawns all enemies, reset player’s hit points and reset the amount of uses on the Flask of Crimson Tears
- Implement a unique ground which can reset the game if the player rests on it

Resettable

The actors (Player, Enemy) and items (Runes, Flask of Crimson Tears) in this UML diagram all implement a Resettable interface. This interface serves to have a function that will be called whenever entities are reset (Interface Segregation Principle). This implementation will allow for us to have the same function but a slightly different reset method for each entity as they require. Also, using the interface will allow for flexibility for any other entity to be added.

Alternatively we could have each resettable entity be reset by the ResetManager. However this goes against the Single Responsibility Principle, in which each class should have one responsibility and Open-Close Principle, where we should not tamper with the existing code given to us.

In the current implementation, the game can reset on two occasions, whenever a player dies, or when the player rests on the Site of Lost Grace.

SiteOfLostGrace

The Site Of Lost Grace extends the abstract ‘Ground’ class. This class will possess the unique ability to reset the game and will have dependency relationships with the ResetManager. This will also have a dependency relationship with Player, in which the Site is dependent on the player to make changes.

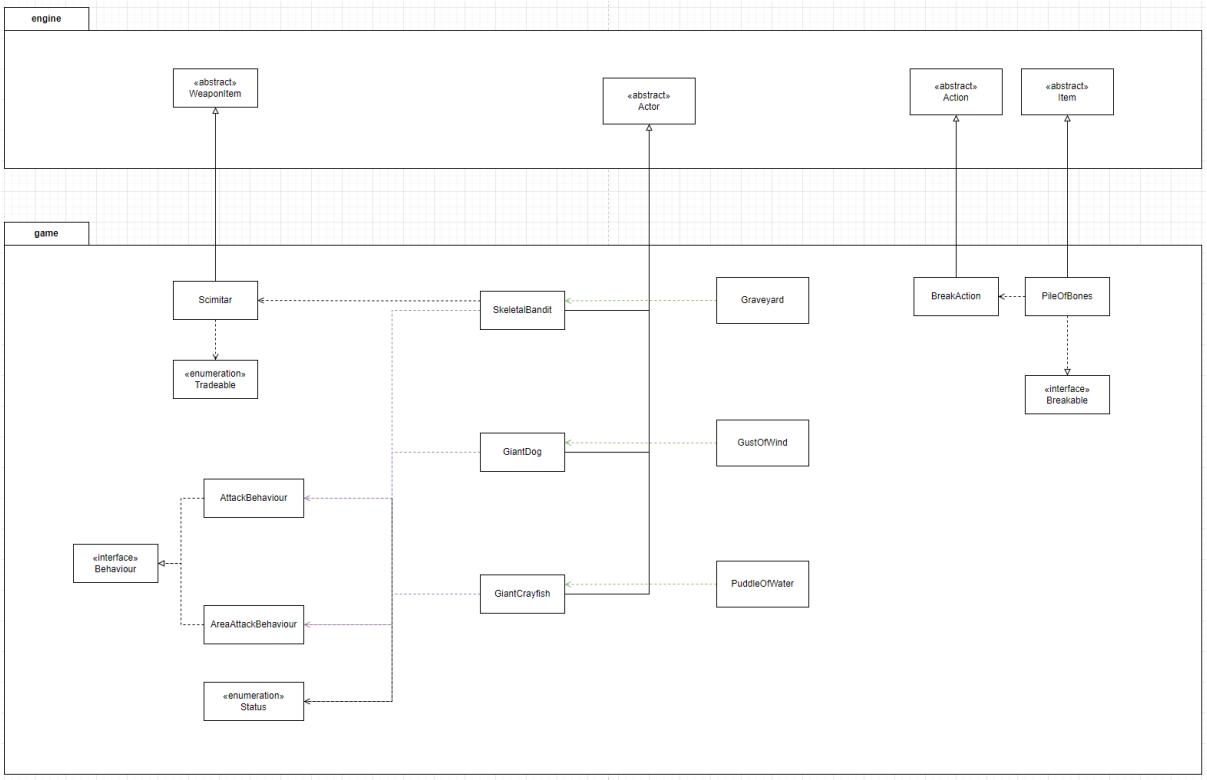
FlaskOfCrimsonTears

Extends the abstract ‘Item’ class. This class has a dependency relationship with Player. The Player is dependent on this item, as the methods in this item are used in Player, and any changes to the item itself will affect the Player using the methods.

Requirement 4

Combat archetypes: Samurai, Bandit and Wretch are the player’s beginning archetype choices. This has a dependency to abstract Player class which inherits another abstract class Actor. Each archetype class will inherit an abstract class called Archetype. Each archetype also begins with a combat weapon, Uchigatana, GreatKnife or Club. Each archetype would inherit its starting weapon, adding it to the player’s inventory.
Combat Weapons: the combat weapons, Uchigatana, GreatKnife and Club all inherit the abstract class WeaponItem, each having unique damage and hitRate attributes. Uchigatana and GreatKnife both have unique skills called Unsheathe and Quickstep respectively. These will be created as classes, and would inherit the abstract class Action. It will then have a dependency relationship to its weapon to perform that Action.

Requirement 5



This diagram represents an object oriented system for the extra enemies to be added to the game.

To avoid repetitions (DRY), a number of inheritance relationships exist in this system, namely:

- **Enemies**: SkeletalBandit, GiantDog, GiantCrayfish are child classes of the Actor class for reasons similar to the enemies in req.1

  SkeletalBandit, GiantDog, GiantCrayfish all exhibit AttackBehaviour and AreaAttackBehaviour as they use intrinsic weapons such as head and giant pincer which perform AOE and targeted attacks, and the Scimatar carried by the Skeletal Bandit, which performs targeted attacks and a special spinning AOE attack. These behaviours decide if an AttackAction or an AreaAttackAction should be performed by the enemy (ie. if there is a target within the enemy's exits, AttackBehaviour/AreaAttackBehaviour will initiate its corresponding attack). This design supports the open close principle as enemies that are added in the future are likely to also have AttackBehaviour and sometimes AreaAttackBehaviour, which can easily be added to the new enemy without having to modify existing code. An alternative implementation could be for AttackBehaviour (and subsequently AttackAction) to handle both targeted and area attacks, however this may violate the single responsibility principle …

  To prevent enemies of the same type from attacking each other (except in area attacks), the Status enum is used to indicate enemies of the same type. This allows our design to be extensible as, if we were to add another enemy that was a canine type, for example, we could add Status.IS_CANINE as a

capability for that new enemy. An alternative implementation would be to use instanceOf() to typecast enemies and check if they're the same type, however this would violate the open close principle as, if were to add another enemy that had the same type, we would have to add another instanceOf() check for this particular enemy, thereby needing to modify the existing code.

- **Scimatar** inherits WeaponItem and similar to the weapons in req.2, uses the Tradeable enum to add Tradeable.SELLABLE and Tradeable.BUYABLE as capabilities, in line with the OCP as mentioned in req.2.
