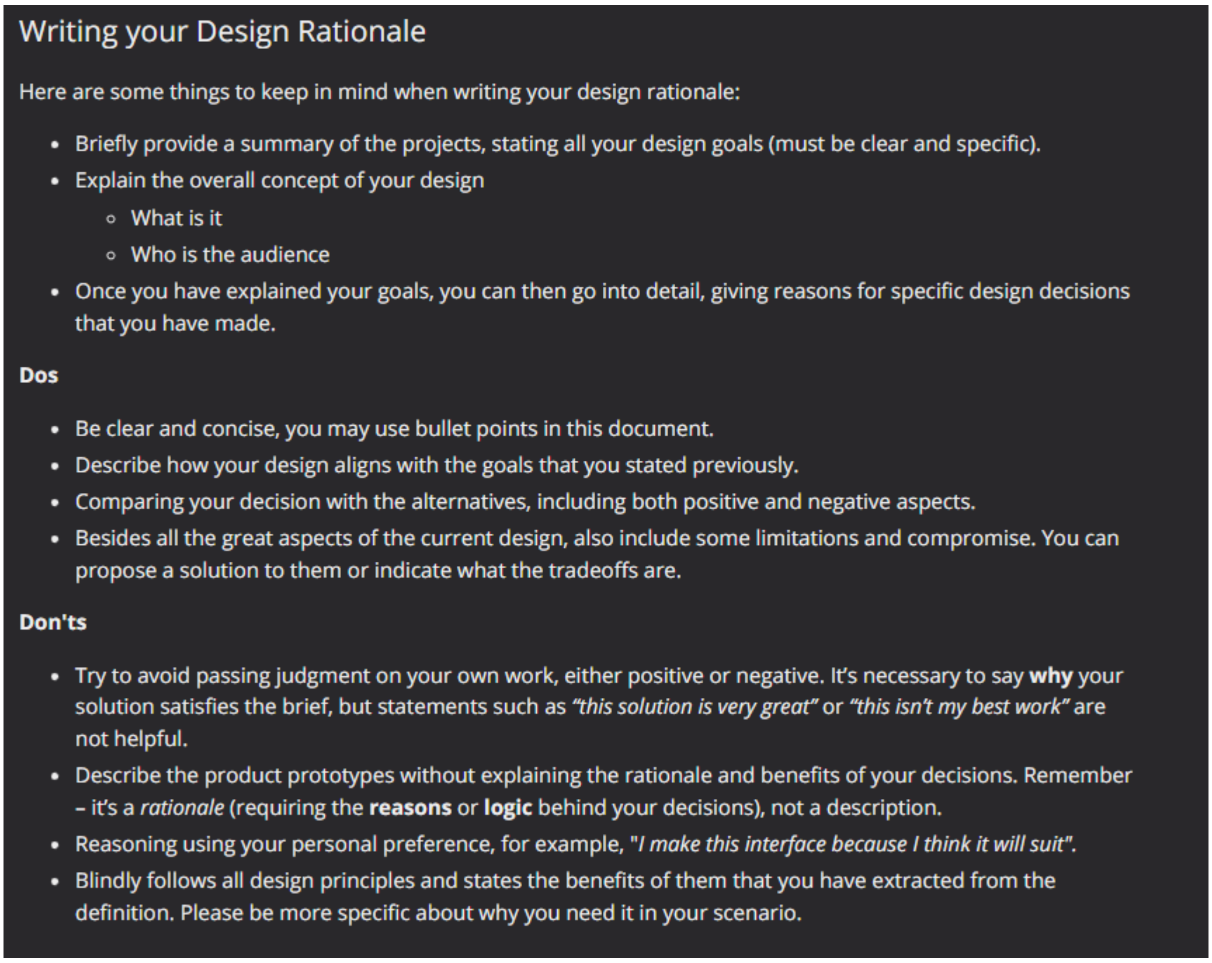
Design rationale: why this approach is good? what's the alternative?

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

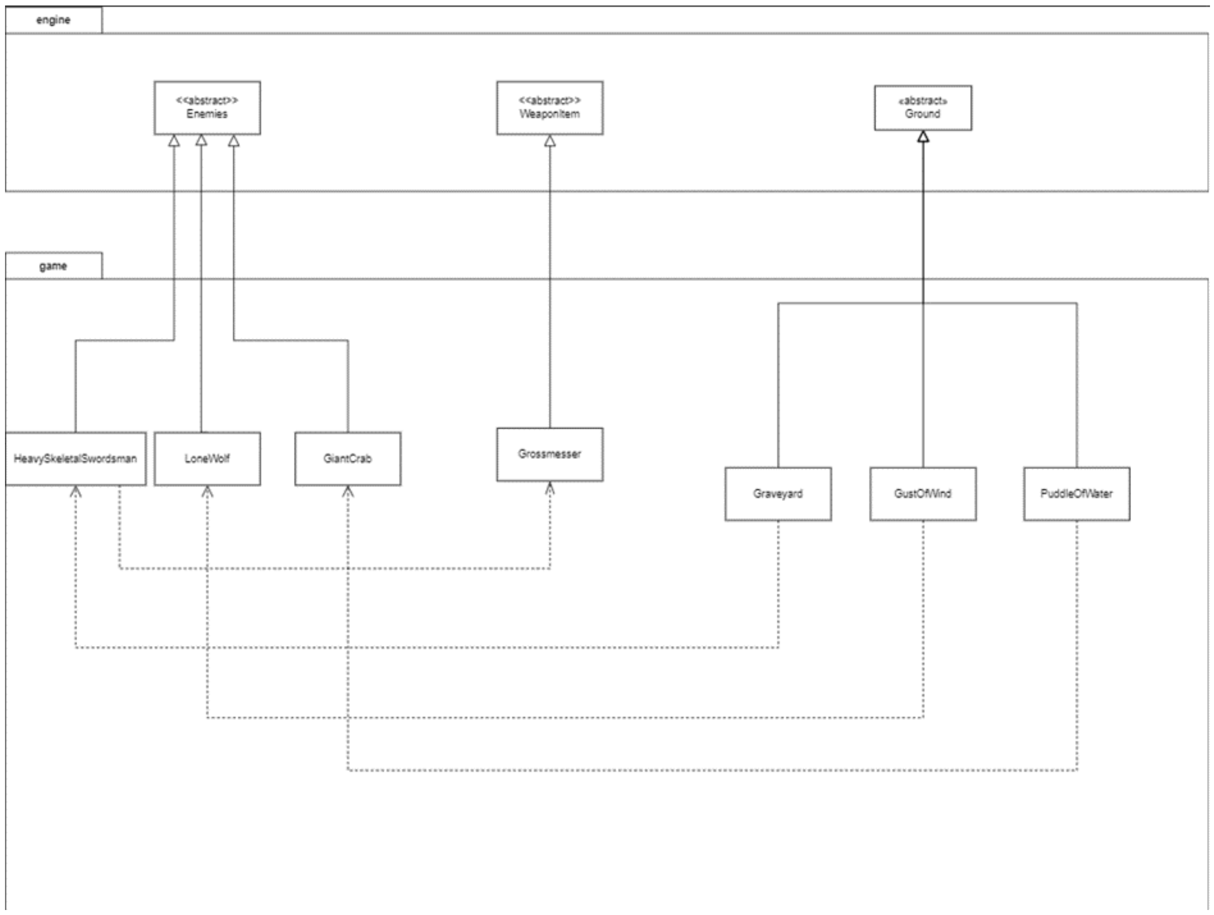
- Be clear and concise, you may use bullet points in this document.
- Describe how your design aligns with the goals that you stated previously.
- Comparing your decision with the alternatives, including both positive and negative aspects.
- Besides all the great aspects of the current design, also include 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



Enemies: Made an abstract class called 'Enemies", this inherits the common properties from actors, and will have extra attributes and methods that are specific and common to the Enemies. We have made the child classes Heavy Skeletal Swordsman, Lone Wolf, and giant crab that inherit the new methods. For this current requirement, each mob interacts with their respective locations as per the requirements via a dependency. 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.

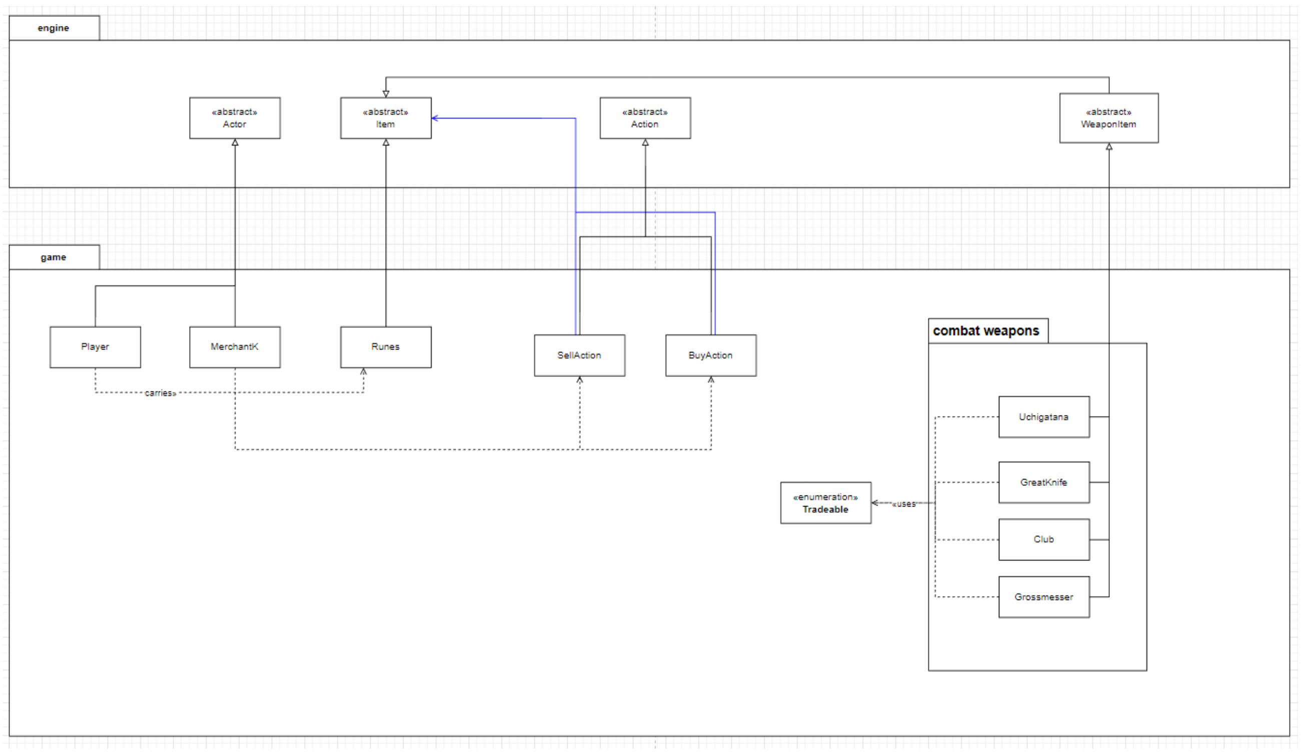
This design overall is good because the important classes and relationships are included in the diagram and follows the DRY principle where the classes which share common attributes extend an abstract 'Enemies'.

Weapons: I have the abstract class 'WeaponItem' which makes use of the Weapons interface from the engine. In this requirement, the weapon Grossmesser made as a child class, and the Skeletal Swordsman has a dependency relationship with this, in which it knows of it and can be affected by it (changes item in inventory, resulting in different attack).

Environments: All three types of environments (Graveyard, Gust of Wind, Puddle of Water) have extended the abstract Ground class. This follows the DRY design principle as the new environments share common attributes and methods described in the abstract Ground class which, when inherited, avoids the need for repetition.

Potential Improvements: This design satisfies the requirements by strictly including all important classes and relationships with the DRY principles. However, it could be made clearer within the diagram for where the abstract classes ‘Enemies’ and ‘WeaponItem’ come/extend from.

Requirement 2



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

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

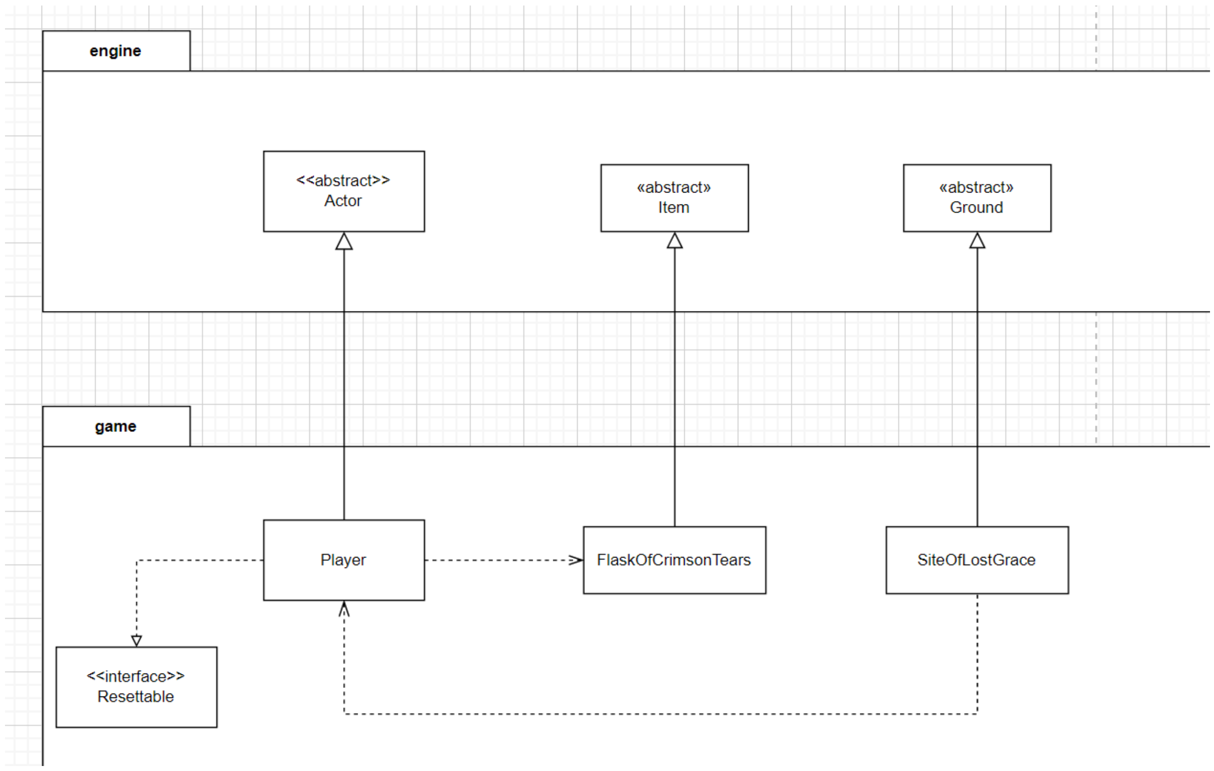
- **MerchantK** extends the Actor abstract class as they share similar methods attributes to other actors (eg. allowable actions, name, display character)
- **Runes** extends the abstract Items class as it shares capabilities similar to other items in the game such as pick up, drop. It would also have 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.
- **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 is used to indicate whether a weapon can be either 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 one of the 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. Hence, an alternative way to 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.

Cons: Merchant K is not an actor that engages in combat, so we wouldn't be using certain methods/attributes such as adding weapon to inventory, hurt, health points. This would be violating ISP?

Requirement 3



Player: Extends the Actor abstract class. Has a dependency relationship with the FlaskOfCrimsonTears and SiteOfLostGrace. These dependencies were discussed in detail below. The player implements the 'Resettable' interface, a class which is implemented through the ResetManager. This is used in Player to indicate whether or not the game should be reset and what is changed/rolled back if the game is reset. This reset can happen under two circumstances, whenever the player dies, or they 'rest' on the Site of Lost Grace.

SiteOfLostGrace: The Site Of Lost Grace extends the abstract 'Ground' class. This has a dependency relationship with Player, in which the Site is dependent on the player for any method to be called or to make changes. This 'Ground' requires the player to be on it so that its methods are called.

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.

Notes: In all of these classes, they have extended an abstract class to inherit common attributes and methods to follow the DRY design principles. By doing so, there is less repetition in code and allows for easier readability.

Things done well:

- Implements the necessary design requirements with relationships

Limitations/ Potential expansions:

- Runes are not included in the UML diagram
- Resettable could have been implemented in SiteOfLostGrace to determine when the game should be reset, however, it does not have the same attributes that the Actors require to reset, thus cannot be done

---

Requirement 4

Requirement 5
