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

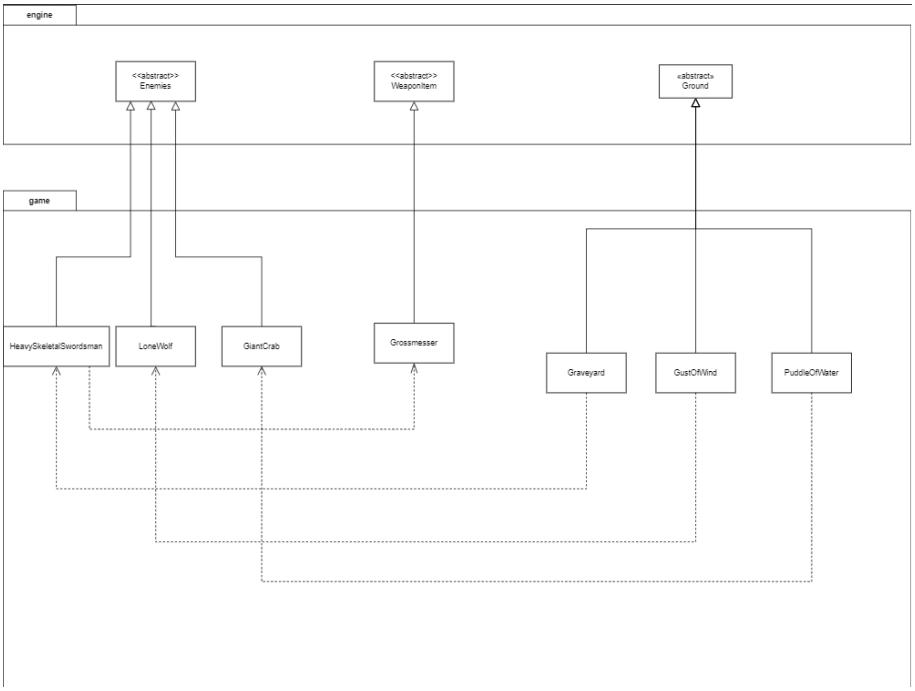
- Be clear and concise, you may use bullet points in this document.
- Describe how your design aligns with the goals that you stated previously.
- Comparing your decision with the alternatives, including both positive and negative aspects.
- Besides all the great aspects of the current design, also include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



Actor: made an abstract class from Actor called ‘Enemies”, this is 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, I made each mob interact 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) pass their respective enemies that they spawn as dependencies into the tick() method.

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).

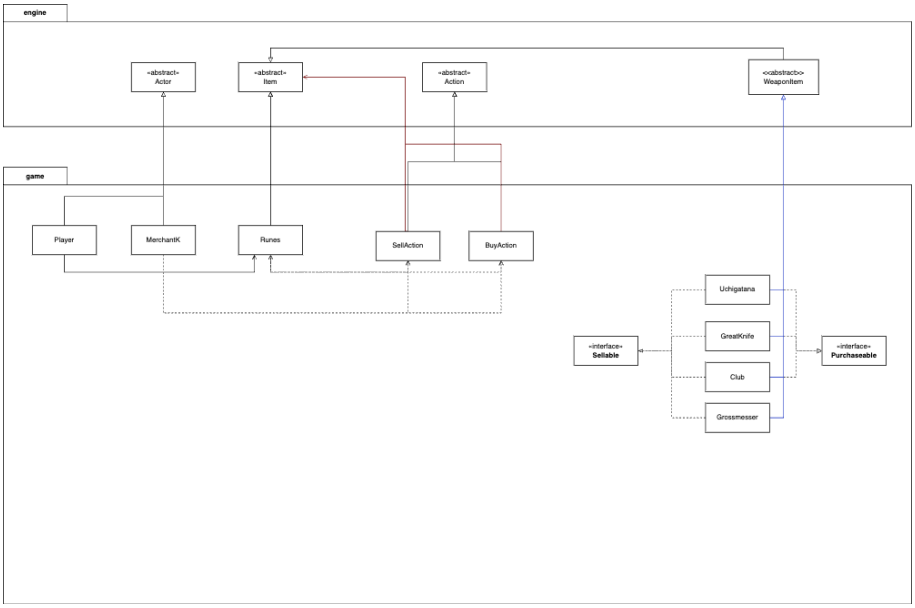
Environments: All three types of environments (Graveyard, Gust of Wind, Puddle of Water) have extended the abstract Ground class. This follows the DRY design principle as the new environments share common attributes and methods described in the abstract Ground class which, when inherited, avoids the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

Requirement 2

Commented [1]: TODO: include comparison with alternative designs, any limitations/compromises + a possible solution/tradeoff for these

Commented [2]: (FROM WHAT I KNOW AND AM READING THIS CAN BE AN ATTRIBUTE/ APART OF IT BUT THEY CAN CONEXIST WITHOUT EACHOTHER, MIGHT JUST BE A NORMAL ASSOCIATION ARROW)

Commented [3]: aggregations are replaced with associations in this unit, though my understanding is the environments have a dependency on the new enemies because they use their tick method to spawn the enemies in their respective environments



MerchantK: MerchantK extends the Actor abstract class

Runes: Runes extends the abstract Items class as it shares actions similar to other items in the game such as pick up, drop, etc, hence following the DRY design principle. Player also forms an association with the Runes class as Runes would be an attribute of Player.

BuyAction/SellAction: To follow the DRY design principles, BuyAction/SellAction extends the Action abstract class as they share similarities with other actions in the game. 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.

Weapons and buyable/sellable interfaces: Uchigatana, GreatKnife, Club and Grossmesser inherit from the WeaponItems abstract class as they all share methods and attributes included in the WeaponItems abstract class. They also implement buyable and/or sellable interfaces to indicate whether they can be bought or sold.

Alternatively, buying and selling 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 user 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. Additionally, having buyable/sellable interfaces allows the system to be extensible as if we were to add more items in the future, we could easily have these items implement buyable/sellable interfaces.

Requirement 3

Requirement 4

Requirement 5
