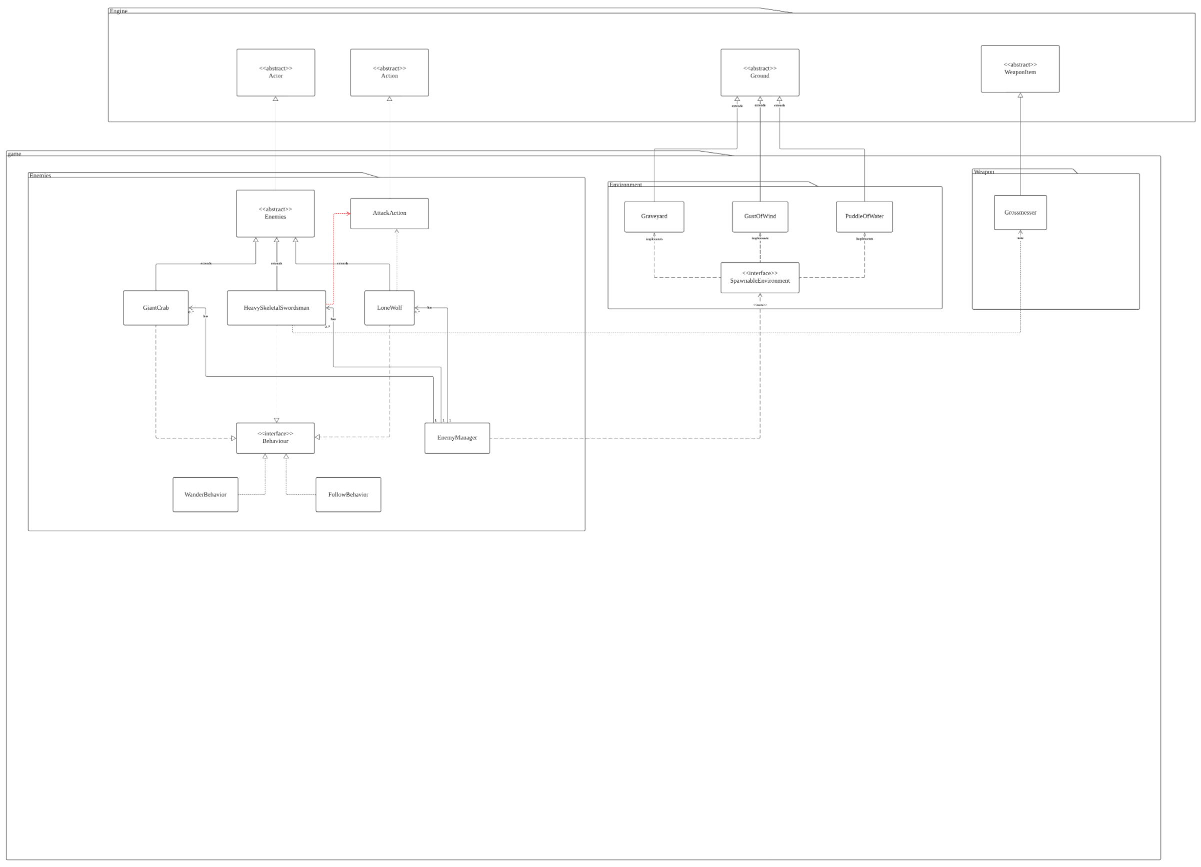
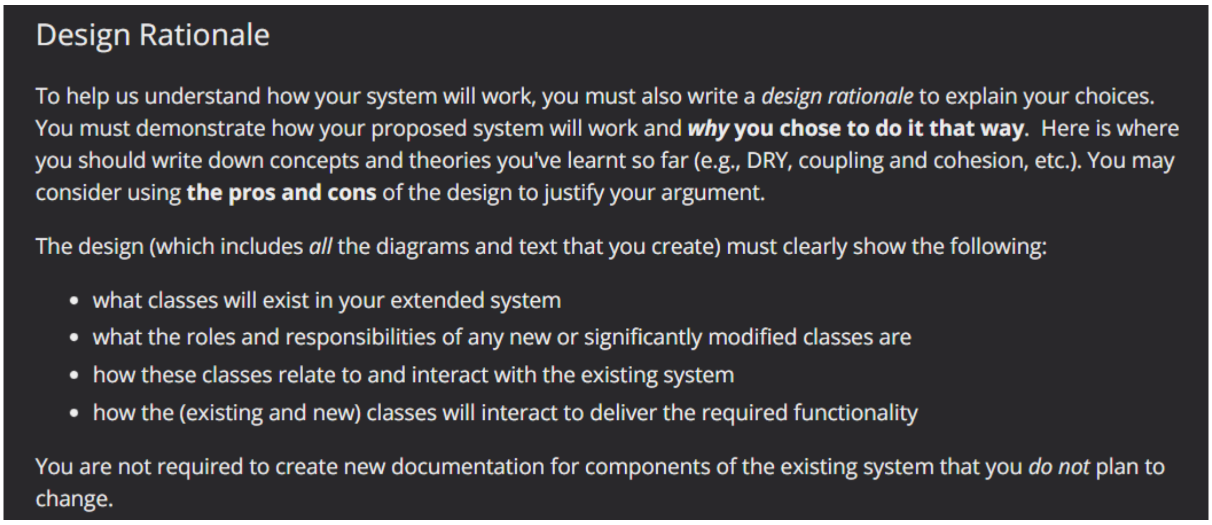
<u>FIT 2099 group 3 assignment 1</u>

Lee Sing Yuan
Loo Li Shen
Yeoh Ming Wei

REQ 1:UML


Engine
<<abstract>> Actor
<<abstract>> Action
<<abstract>> Ground
<<abstract>> WeaponItem
extends
extends
extends
game
Enemies
<<abstract>> Enemies
AttackAction
extends
extends
extends
GiantCrab
HeavySkeletalSwordsman
LoneWolf
has
has
has
0..*
0..*
0..*
1
1
1
<<interface>> Behaviour
EnemyManager
WanderBehavior
FollowBehavior
Environment
Graveyard
GustOfWind
PuddleOfWater
implements
implements
implements
<<interface>> SpawnableEnvironment
<<uses>>
Weapon
Grossmesser
uses


Design Rationale

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.

Goal:
How was it implemented:
Why this method?:
Alternative:
    Pro:
    Con:

REQ 1:Rationale thinking of adding interface for AOE skills

1.A) SpawnableEnvironment

The **goal** of this interface is to reduce amount of inheritance and dependencies. It is **implemented** by all the environments such as Graveyard (environments that can spawn enemies). The **reason** for this method is because there is already a ground parent class and in the future there could be more environments that could spawn enemies. The **alternative** to this method is to create another parent class. **By doing so**, we could apply DRY. However, this causes multiple levels of inheritance and will make it hard to maintain and debug.

1.B) enemy manager

Goal: Reducing dependency
How was it implemented: EnemyManager class ,handles arraylist of enemies, spawn and despawns. (for example 1 spawn method and then if conditions to select which enemy to spawn or despawn rather then interface [alternative method])
Why this method?: Because with this it will be easier to spawn and despawn enemies in the future.
Alternative: 1 enemy connect to one environment (something like interface to spawn enemies)
    Pro: more direct and easier to debug
    Con: hard to manage all enemies since they are all in different classes , a lot of code repetition (for example : in Graveyard , need to have/ implement method which is quite similar to other environments like Puddle but with minor tweeks. Breaks DRY)

1.B.1 and 1.B.3) skeleton skill and crab attack area is coded in class

Goal: achieve DRY
How was it implemented: code the skill in the class and if there is a need later like in **REQ 5**, make this class a parent
Why this method?: Because the skills are unique to them and for example in **REQ 5** the code is exactly the same with no changes (based on the specs at the current time)
Alternative: interface for the skills
    Pro: easy to implement (copy paste)
    Con: repeated code , breaks DRY

1.C) Grossmaseer skill

Goal: Reduce the dependency and need of other classes and interfaces
How was it implemented: coding the skill inside the class
Why this method?: Because it reduces the amount of code
Alternative:

Pro:

Con:

REQ 2: Rationale

2.A) handling runes

Goal: to reduce the amount of dependency and to ensure that only actors who can collect runes can collect runes and the same for dropping runes
How was it implemented: having a rune manager class that increases and decreases the amount of runes and 2 interfaces CollectRune and DropRune
Why this method?: because it reduces the amount of dependency
Alternative: directly using runes in the enemies abstract class and player class

> **Pro:** reduces a lot of code and associations and dependency
> **Con:** duplicate code

2.B) Trader
The current design of the Trader class is built upon the SOLID principle of Dependency Inversion by having the class depend on abstract classes instead of concrete implementations. This makes the Trader class more flexible and extensible, allowing for new types of weapons or inventory systems to be added in the future without affecting the Trader class.

2.C) Creating weapons
Inheritance and abstraction are used to create a more modular and manageable architecture. The classes "<<abstract>> Weapon" and "<<abstract>> Inventory" offer a framework for future additions of new weapons and inventories, while the current classes can be altered to include new functionality. With this design strategy, which adheres to the Open-Closed SOLID concept, the system may be modified without changing the existing code.

To maintain low coupling and high cohesion, the "Trader" class is responsible for handling trading and the "Weapon" classes are responsible for defining their respective weapons' properties and methods. The "Inventory" class is responsible for managing the player's inventory. This separation of responsibilities makes it easier to maintain and modify the system in the future.