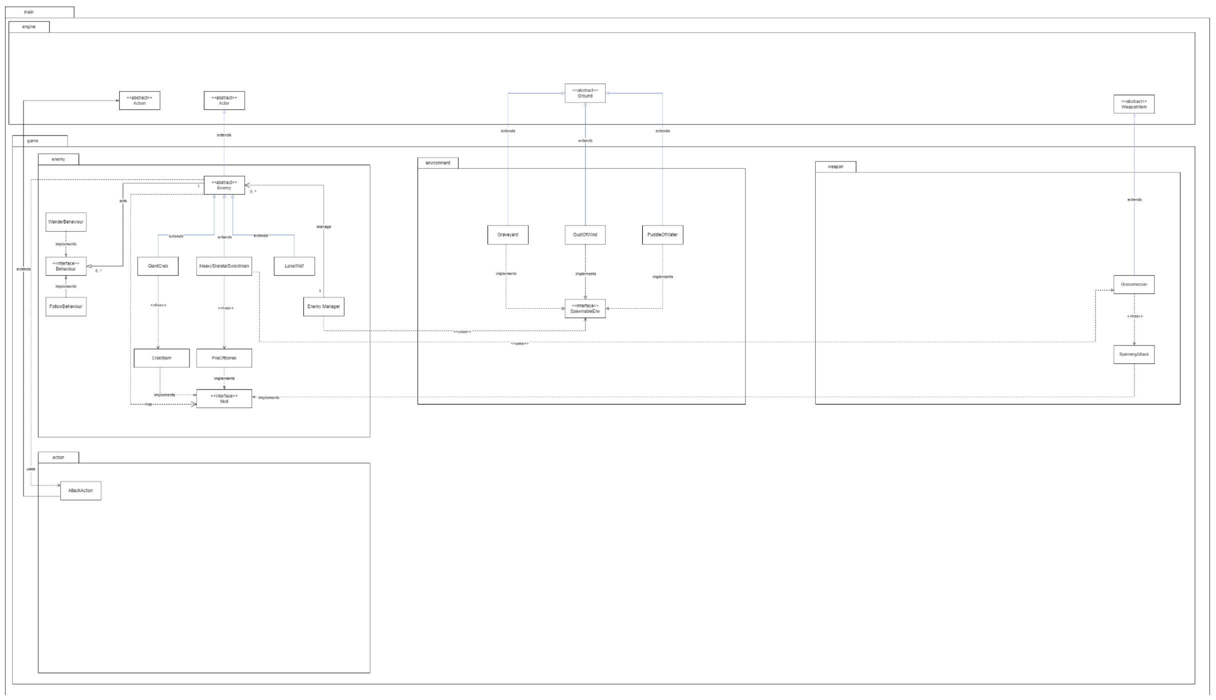
<u>FIT 2099 group 3 assignment 1</u>

Lee Sing Yuan
Loo Li Shen
Yeoh Ming Wei

REQ 1:UML


main
engine
<<abstract>> Action
<<abstract>> Actor
<<abstract>> Ground
<<abstract>> WeaponItem
extends
game
enemy
<<abstract>> Enemy
1
0..*
acts
WanderBehaviour
implements
<<interface>> Behaviour
0..*
implements
FollowBehaviour
extends
extends
extends
extends
GiantCrab
HeavySkeletalSwordman
LoneWolf
manage
1
Enemy Manager
<<has>>
<<has>>
CrabSlam
PileOfBones
implements
implements
has
<<interface>> Skill
implements
environment
extends
extends
extends
Graveyard
GustOfWind
PuddleOfWater
implements
implements
implements
<<interface>> SpawnableEnv
<<uses>>
<<uses>>
weapon
extends
Grossmesser
<<has>>
SpinningAttack
action
uses
AttackAction

REQ 1:Rationale thinking of adding interface for AOE skills

1.A)

SpawnableEnvironment

The **goal** of this interface is to reduce the 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

The **goal** of this is to adhere to the DRY principle as many of the enemies have a lot of similarities. It was **implemented** by inheriting the actor class and adding a few extensions and an association to behaviour.

Enemy skills

For all the skills present in the game, it would **implement** the interface Skill. The **objective** of this is to somehow relate all the skills so that when attacking we can see use the skills as a data type. This will be **implemented** by having the enemy parent class have an ArrayList<Skill>. For enemies without skill, they will have the ArrayList of len 0. Then, the enemy skills like “PileOfBones” will be used in the HeavySkeletonSwordsman as a local param to initiate the skill (like change the skeleton’s display to X). This makes it easier to maintain the skills.

The alternative to this method was to have the skills coded in the classes (example: Pile of Bones in Heavy Skeleton Swordsman).

EnemyManager

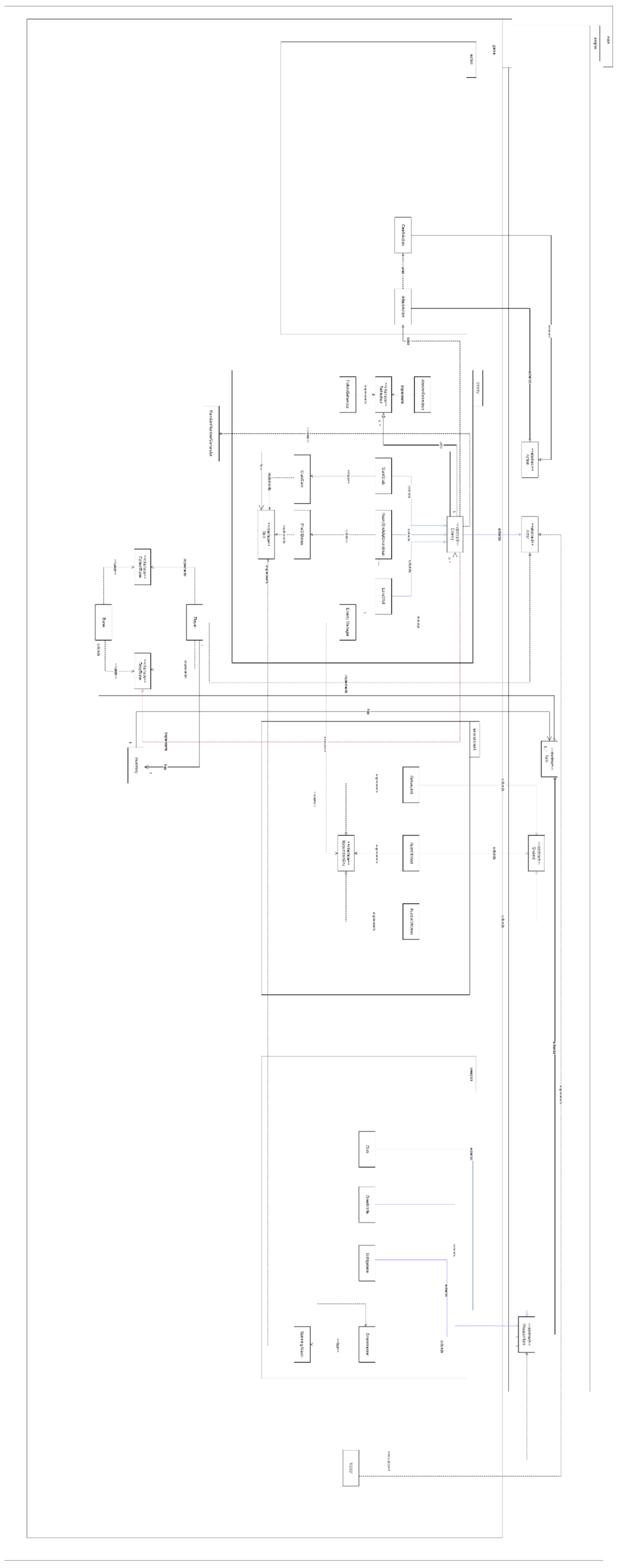
The **objective** of this enemyManager is to manage the arraylist of enemies, spawning and despawning of enemies. It will be implemented like this for example: 1 spawn method and then “if” conditions to select which enemy to spawn or despawn. This method is easier and adheres to SOLID principles which makes it easier to maintain and improve. The alternative method would be to directly have an association to the enemy for that environment. For example: Skeleton spawns in graveyards, thus graveyard handles spawning of skeletons and despawning of skeletons. This method is very direct but violates SOLID principles and hard to maintain. Besides that there would also be a lot of code repetition which violates the DRY principle.

1.C)

Grossmasser skill

The Grossmasser has a skill called spinning attack. As stated before with the enemy skills, the skill will have another class which will implement the Interface skill to have a correlation.

REQ 2: UML



REQ 2: Rationale

2.A)

rune class , drop rune interface and collect rune interface

The **goal** of the rune class is to be able to drop and collect runes. It is **implemented** by considering runes as items without capabilities. WE will ensure that runes are never able to add or remove capabilities by overriding the methods.

For enemies,
Enemies will implement drop runes and get a random amount of rune upon instantiation of the RUNE class

FOR player
Player will implement both drop and collect rune.

The alternative to this method was to have runes be represented by an integer. The issue with this method is how would we display the runes when dropped.

2.B)

Trader

The **purpose** of the trader class is to handle selling and buying of weapons. Since, the trader can be considered an actor, it inherits the actor class.

Trading

Selling or buying weapons, will be implemented using an if condition. For example: if this weapon is sellable, then sell else don’t sell. This method follows the concept of **open close principle**, as in if a new type of weapon exist and its sellable but not buyable, we don't have to modify the code in the trader because we just have to set the attributes of the weapons to be accordingly.

2.C)

Creating weapons

All weapons will inherit from weaponItem **to apply the DRY principle.**

Storing weapons

The player will be associated with a class called Inventory which acts as an itemManager. The **goal** is to separate the player from managing the inventory(SRP) .The inventory will be **implemented** to have an ArrayList<Item> and to add and remove items from the inventory.