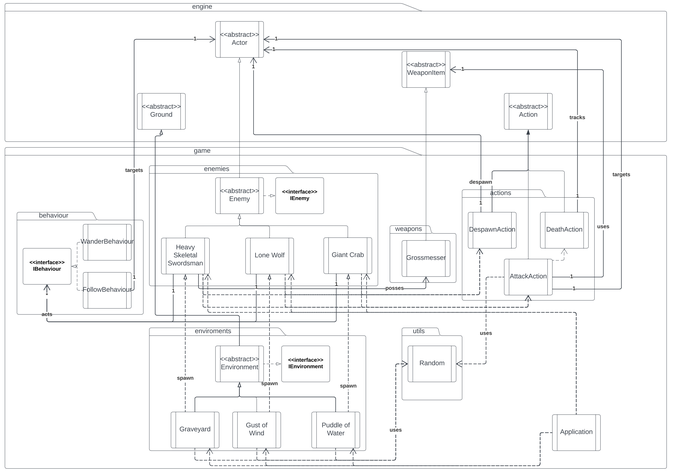
REQ1: Environments & Enemies



Design Rationale for REQ1

During the designing of req1, serveral concept and theorties is taken into account to ensure a good OOP design, and does not change anything in engine package.
Inside the game package of this uml, contains a total of 6 package and 17 concrete classes , 2 abstract class, 3 interface.

classes are designed to follow single resposibility princible, for example:
In enemy package, Enemy classe is reponsible for enemies general attributes. HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes are responsible for their type of enemy attributes. same goes with other classes.

Also designed to follow interface segregation principle, for example:
Interface is implemented for enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes), enviroment(Graveyard, GustOfWind, and PuddleOfWater classes), and behaviour, because those class have similarities. this improve modularity and maintainability, and easier for creating new child class by implementing interfaces.

designed to follow Liskov Substitution Principle, for example:
enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes) and enviroment(Graveyard, GustOfWind, and PuddleOfWater classes), those child class can replace their parent class and still keep the program working.

designed to follow Dependency Inversion Principle, for example:
enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes) is depending on IBehaviour interface instead of a concrete class, which reduces coupling between enemies and other concrete class and making it more flexible and adaptable to changes, extends. same goes with actions.

designed to follow open and close Principle, for example:
Weaponitem class attack a single actor, while Grossmesser are overrided to attack multiple target, which makes Weaponitem class open for extension but closed for modification. this way the code is easier to maintain and test.

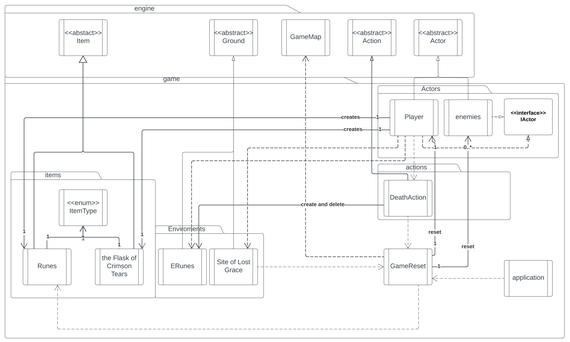
Inheritance is used in enemies, enviroments, weapons and more, which reduce code duplication and improve reusablitity of the code base. Making it easier to add new enemies, enviroments, actions and weapons in the future.

Multiple package is created to improve cohesion and coupling, making it easier to manage, update and understand.

PileOfBones is not implemented as a class because I believe it is better to implemented within the HeavySkeletalSwordsman class since PileOfBones doesn't require much code and seems easier to implenment into the HeavySkeletalSwordsman class.

In conclusion, the design ensure the program to be well structured, easyy to maintain, scalable and a good OOP design.

REQ3: Grace & Game Reset



Design Rationale for REQ3

During the designing of req3, serveral concept and theorties is taken into account to ensure a good OOP design, and does not change anything in engine package.
Inside the game package of this uml, contains a total of 4 package and 10 concrete classes , 0 abstract class, 1 interface.

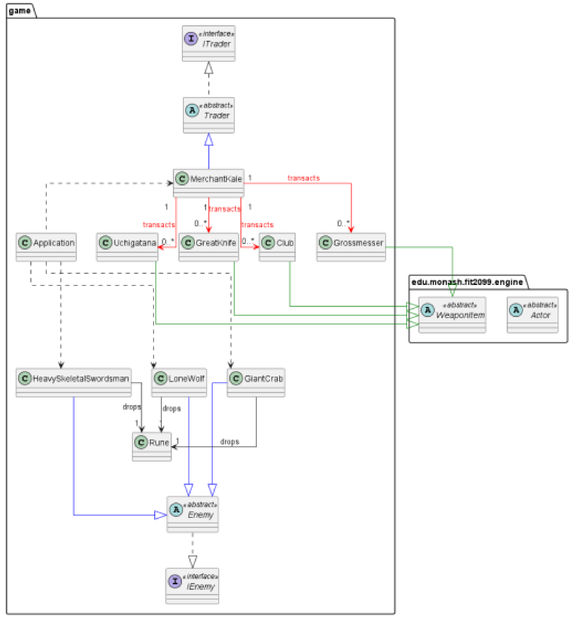
classes are designed to follow single resposibility princible, for example:
Runes class used to keep track of player's rune, ERune is used to put the runes on the ground.

Also designed to follow interface segregation principle, for example:
Interface is implemented for player and enemies, because those class both have health and drop runes when die. this improve modularity and maintainability, and easier for creating new child class by implementing interfaces.

Inheritance is used in enemies, enviroments and items, which reduce code duplication and improve reusablitity of the code base. Making it easier to add new enemies, enviroments, actions and weapons in the future.

Multiple package is created to improve cohesion and coupling, making it easier to manage, update and understand.

In conclusion, the design ensure the program to be well structured, easyy to maintain, scalable and a good OOP design.



REQ2 Design Rationale

The idea was to implement a trader (Merchant Kale) who the player can buy and sell weapons using Runes. A Rune will be dropped when an enemy is killed by the player, when picked up the player will receive a certain number of Runes depending on the enemy. In this UML Diagram there are 10 classes (7 new), 3 abstract classes (two of which are from the engine), and 1 interface.

Single Responsibility: The Trader class is a parent of the various Traders that will be added in the future. For now, only MerchantKale is a child of this class. The Trader class will be implemented to follow the single responsibility principle. This class is solely responsible for the various trader's attributes and methods. This is also the same for the Enemy class implemented in REQ1.

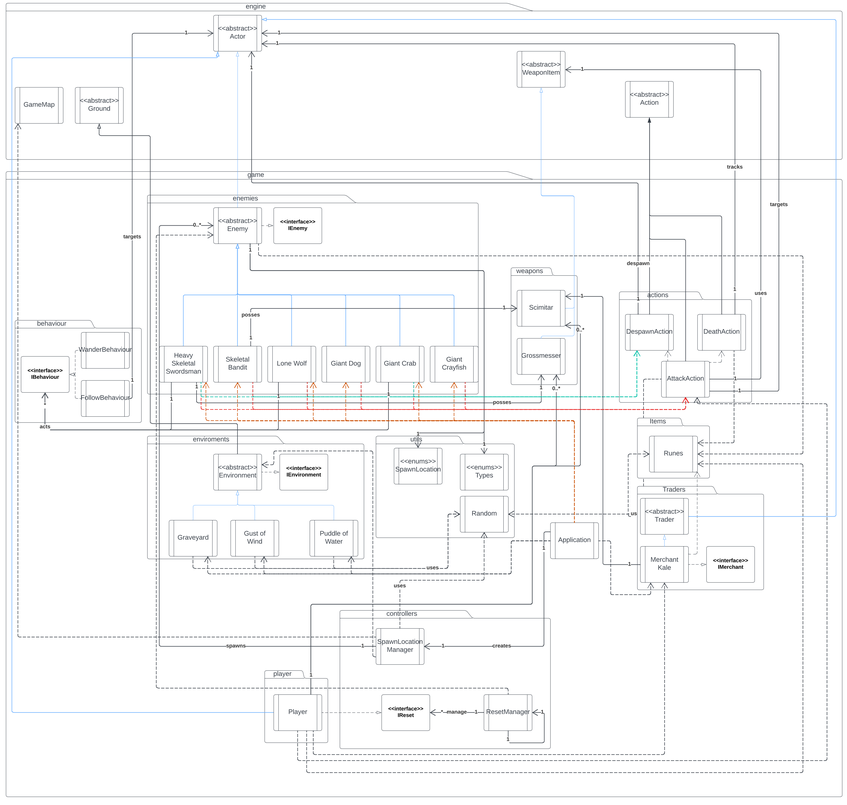
Interface Segregation: The interface ITrader was added to follow the interface segregation principle.

Liskov Substitution: All future and current traders will be replaceable by their child classes and still keep the program functioning.

Dependency Inversion: The Trader's depend on their interface instead of a concrete class, which minimises the dependencies for them. This will make the program more flexible and we can change more.

My initial thought was to drop multiple Rune objects when an Enemy is killed by the player. But this would not be optimal, and we are limited by our game interface. Instead, One Rune object will be dropped onto the map and that will store the number of Runes as a variable. This Rune can be picked up by the player.

REQ5: More Enemies (HD requirement)



Design Rationale for REQ5

Req5 design is similiar to Req1 design because they have similiar classes, serveral concept and theorties is taken into account to ensure a good OOP design, and does not change anything in engine package.
Inside the game package of this uml, 4 new package and 11 new concrete classes , 2 abstract class, 2 interface.

Since REQ5 is similiar to REQ1 the same concepts, theoies, principle mentioned in req1 design rationale is also implemented in req5 with same example, check out Req1 for more information on the concepts, theories princible followed by REQ5 UML.

Enum types class is created to ensure the same type won't target each other. this won't need to be done in req 1 because req 1 doesn't have two class of the same type.

enum spawnlocation class is created to help spawnlocation manager class to spawn the enemies in the right location. spawnlocation manager depends the GameMap to determine east and west of the map.

Runes class is created to manage player's runes and transaction when trade or defeat an enemy.

The new child class of enemy of work the same as enemies in REQ1 and same goes with the new weapon. but the new weapon can be sold to the merchant, since a buy back option wasn't mentioned so once the weapon is sold it is deleted from the player inventory.

merchant class is child class abstract trader class which is child class of actor. merchant has a Scimitar for sell so when ever players buys the weapon it is copy to player inventory which means merchant only need one scimitar.

The new package created help with readability of the UML, etc, similiar to what mentioned in req1.

classes are designed to follow single resposibility princible, for example:
In enemy package, Enemy classe is reponsible for enemies general attributes. HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes are responsible for their type of enemy attributes. same goes with other classes.

Also designed to follow interface segregation principle, for example:
Interface is implemented for enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes), enviroment(Graveyard, GustOfWind, and PuddleOfWater classes), and behaviour, because those class have similarities. this improve modularity and maintainability, and easier for creating new child class by implementing interfaces.

designed to follow Liskov Substitution Principle, for example:
enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes) and enviroment(Graveyard, GustOfWind, and PuddleOfWater classes), those child class can replace their parent class and still keep the program working.

designed to follow Dependency Inversion Principle, for example:
enemy(HeavySkeletalSwordsman, LoneWolf, and GiantCrab classes) is depending on IBehaviour interface instead of a concrete class, which reduces coupling between enemies and other concrete class and making it more flexible and adaptable to changes, extends. same goes with actions and SpawnLocationManager.

designed to follow open and close Principle, for example:
Weaponitem class attack a single actor, while Grossmesser are overrided to attack multiple target and Scimitar is overrided to attack multi target or single target, which makes WeaponItem class open for extension but closed for modification. this way the code is easier to maintain and test.

Inheritance is used in enemies, enviroments, weapons, player, trader and more, which reduce code duplication and improve reusablitity of the code base. Making it easier to add new enemies, enviroments, actions and weapons in the future.

Multiple package is created to improve cohesion and coupling, making it easier to manage, update and understand.

PileOfBones is not implemented as a class because I believe it is better to implemented within the HeavySkeletalSwordsman class since PileOfBones doesn't require much code and seems easier to implenment into the HeavySkeletalSwordsman class.

In conclusion, the design ensure the program to be well structured, easyy to maintain, scalable and a good OOP design.