FIT2099 Object-oriented Design and Implementation

Assignment 2 - UML Diagrams, Interaction Diagrams & Design rationale

Group members:

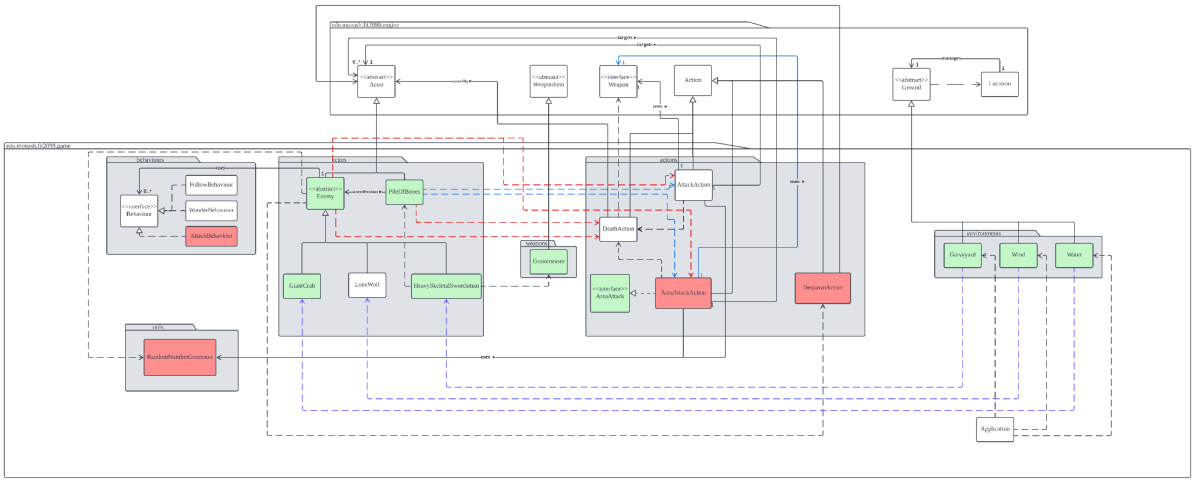
(1) Muhammad Mustafa Khan 33041202
(2) Nurin Damia Ahmad Azhar 32903510
(3) Iliyana Samsudin 32738781

Link to Contribution log:
https://docs.google.com/spreadsheets/d/1e7polQC9IYCEiwOh9_jiUQ6UL1qxGAKn3CofYyGJVig/edit#gid=0

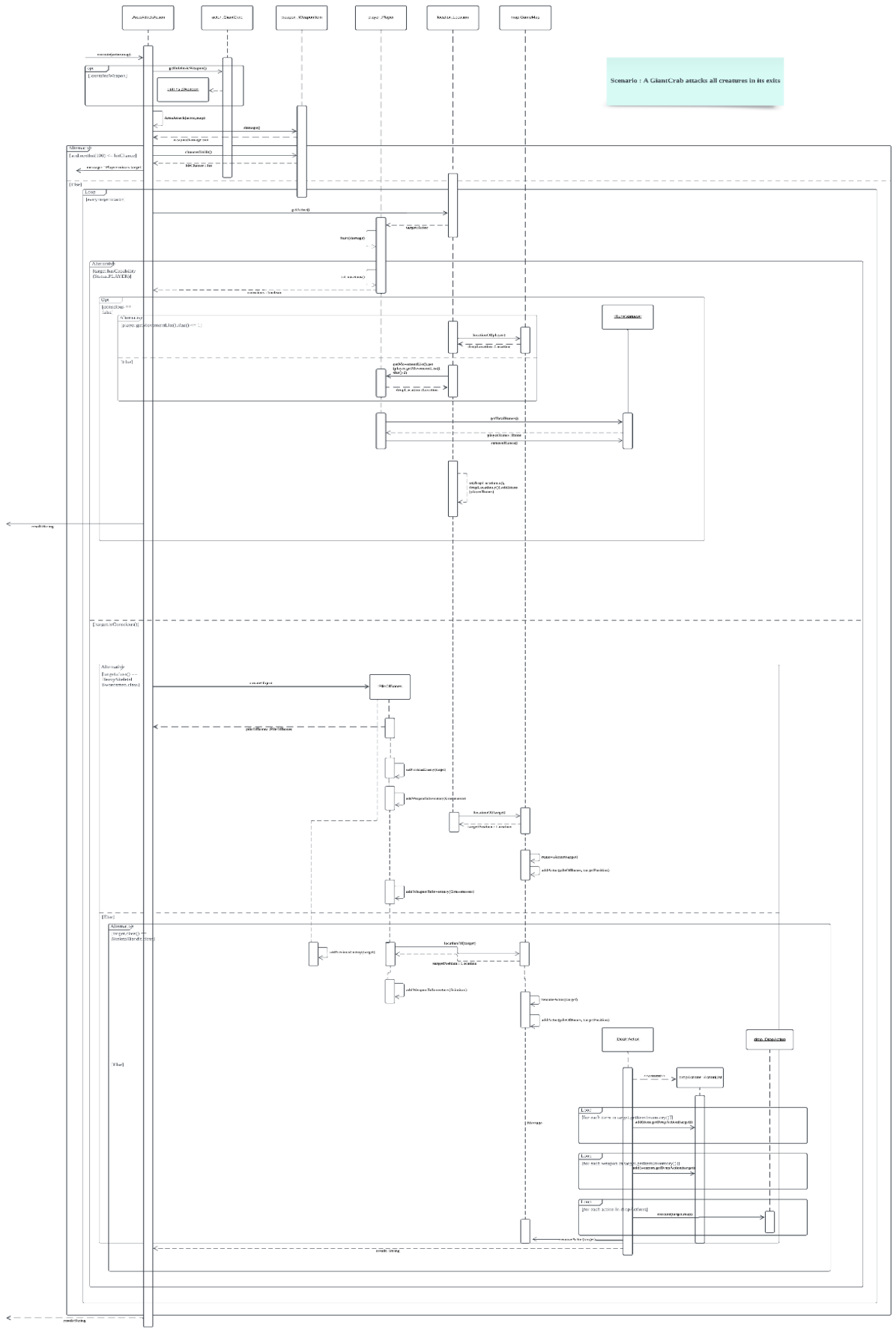
Note: Red classes are the newly added classes for assignment 2.

Requirement 1 - Environment & Enemies

UML Diagram



Interaction Diagram


Scenario : A GiantCrab attacks all creatures in its exits

Design Rationale

The UML diagram above represents the object-oriented system of the Environment and Enemies that are available in the game.

Graveyard, **Wind**, **Water** are 3 concrete classes that are extended from the abstract class Ground. All of the 3 types of environment are extended from the Ground abstract class because they share common attributes and methods which a type of a Ground should have. However, each environment is able to have unique attributes and methods such as unique characters and different probability to spawn different types of enemies. Hence, creating 3 subclasses allows us to cater to each class uniqueness and reduce code repetition which follows the DRY principle (Don't Repeat Yourself).

Enemy class is an abstract class and is extended from the Actor abstract class which provides all the functionalities required by an Actor to create NPC (non-playable character). Then, **HeavySkeletalSwordsman**, **LoneWolf**, **GiantCrab** are concrete classes that are extended from the Enemy class. Although there is a multiple tier of inheritance, each of the enemies share common attributes and methods such as the **AttackBehaviour**, **WanderBehaviour**, **FollowBehaviour** which are already added in the Enemy class. The despawning of enemies is done in the Enemy class by returning a new **DespawnAction**. By extending from the Enemy class, it allows us to reduce code repetition which follows the DRY principle (Don't Repeat Yourself). Any new additional attributes and behaviours can be implemented in its respective classes to cater each class uniqueness.

A new **AttackBehaviour** class that implements the behaviour interface is added. This class is used to figure out an attack action that will be invoked by the enemy. The downside of this is that whenever a new type of attack action is added for another enemy, the execute() method will be extended and modified, violating the Open-Closed principle.

The **DespawnAction** is a class that is an action class that is used to despawn an actor from the map. The benefit of this is that all enemies can be removed from the map by instantiating this action. This allows us to reduce code repetition.

PileOfBones is a class that is extended from the Actor abstract class which provides all the functionalities required by an Actor. This class is responsible for the Heavy Skeletal Swordsman to turn into a pile of bones when it is attacked. The PileOfBones class will be created at the location where the Heavy Skeletal Swordsman was attacked and will be removed from the map if it is destroyed and drop the Grossmesser, else a new Heavy Skeletal Swordsman will spawn after 3 turns. By creating a PileOfBones class, it allows the Heavy Skeletal Swordsman class or other classes that require it by just instantiating it instead of implementing it in each class. The PileOfBones class stores the previous enemy before it turns into a pile of bones, allowing them to re-spawn their previous enemy if it is not destroyed. Hence, this shows that the benefits of the PileOfBones class allows us to follow the SRP (Single Responsibility Principle) which states that a class should have only one responsibility and it also helps in reducing code repetition. It also prevents the WET (Wasting Everyone's Time) principle. However, by not making the PileOfBones class a subclass of Enemy, we will have repeating codes in other classes to cater for the pile of bones.

Furthermore, a new status is created for the **LoneWolf** class to add as a capability in the constructor. This allows the lone wolf to attack other enemies but not its own kind.This also applies to the **GiantCrab** and **HeavySkeletalSwordsman**. The **AreaAttackAction** class implements the **AreaAttack** interface and extends the Action class. Both spinning attack for the Grossmesser and the slam attack action for the crab's special attack behaves the same way, where every actor surrounding the attacker will be dealt with damage. The only difference is that one uses a weapon item, and one doesn't. So we have to create two constructors for the AreaAttackAction class to deal with that. By implementing only AreaAttakAction instead of one SlamAttackAction class and one SpinningAttackAction class, we reduced the code repetition greatly as they have the exact same properties.

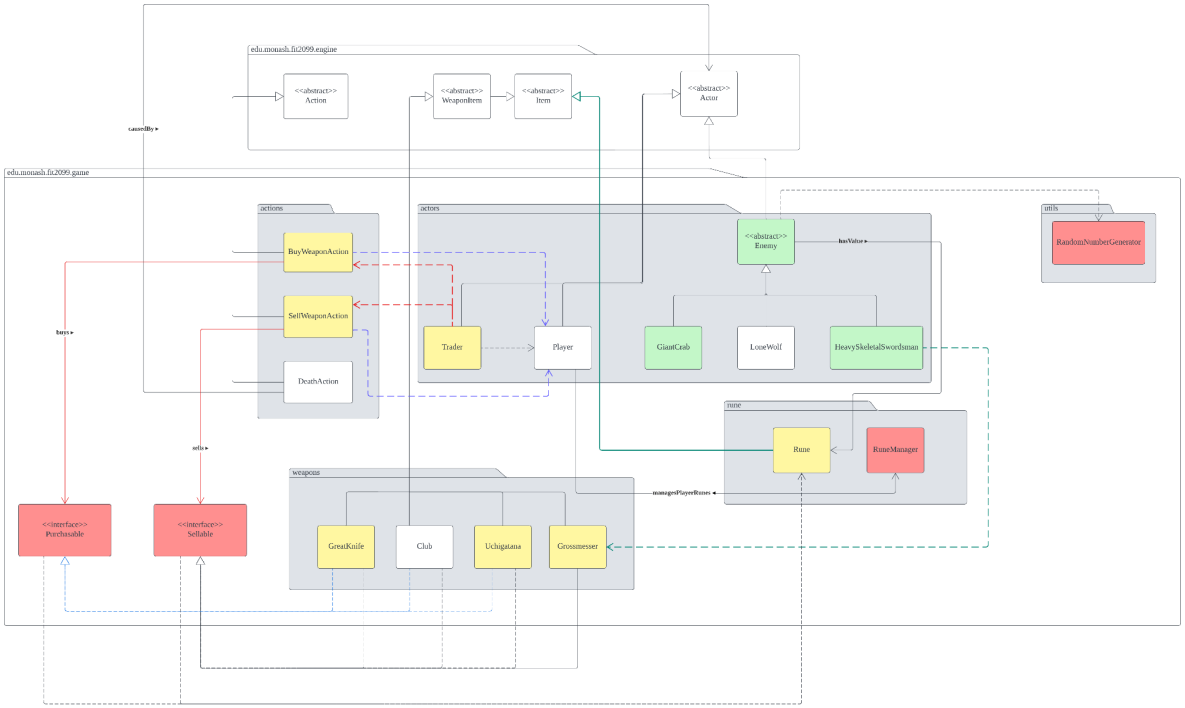
Lastly, **Grossmesser** is a weapon that is used by the Heavy Skeletal Swordsman. It is extended from the WeaponItem class which extends from the Item abstract class and implements the Weapon interface. This allows new Weapon types to be created without having to modify the existing code. By extending the WeaponItem class, Grossmesser shares common attributes and methods that are implemented in the WeaponItem abstract class and the Item abstract class which can be reused. This allows us to follow the DRY (Don't Repeat Yourself) principle.

How this design can be extended in the future:

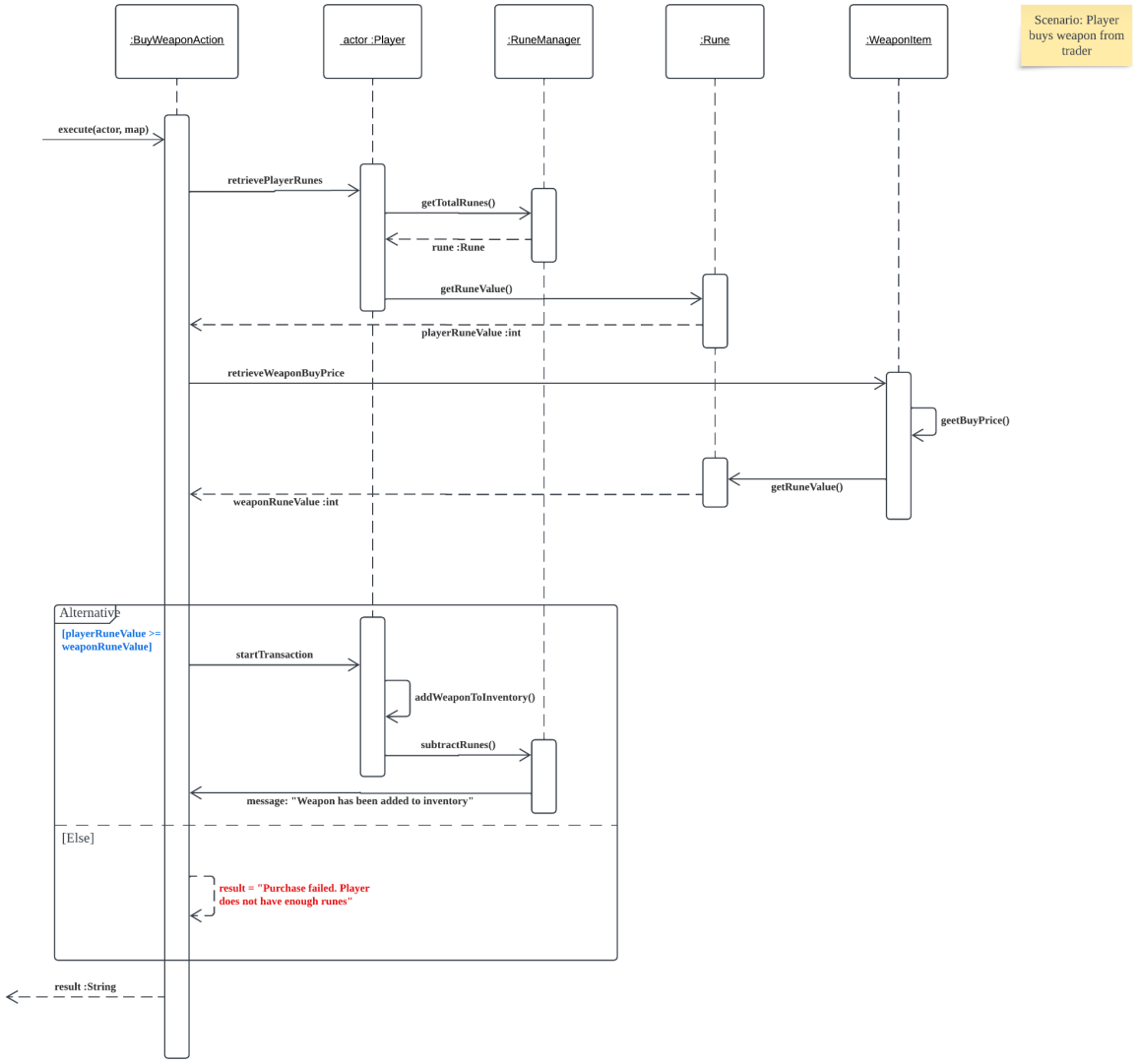
- New enemies added can just extend from the Enemy class. In their own class, only the constructor and other additional characteristics of the enemy will be added. For example the new enemies that will be added for requirement 5.
- Any new attack actions added will be added into the attack behaviour class, but with modifications to the code.
- Skeletal types of enemies that can turn into a pile of bones will have to just add a new capability, Status.SKELETAL_TYPE, so that they can turn into bones after hit point becomes 0.

Requirement 2 - Trader & Runes

UML Diagram


edu.monash.fit2099.engine
<<abstract>>
Action
<<abstract>>
WeaponItem
<<abstract>>
Item
<<abstract>>
Actor
causedBy
edu.monash.fit2099.game
actions
BuyWeaponAction
SellWeaponAction
DeathAction
actors
<<abstract>>
Enemy
hasValue
Trader
Player
GiantCrab
LoneWolf
HeavySkeletalSwordsman
utils
RandomNumberGenerator
rune
Rune
RuneManager
buys
sells
weapons
managedPlayerRunes
<<interface>>
Purchasable
<<interface>>
Sellable
GreatKnife
Club
Uchigatana
Grossmesser

Interaction Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the Trader and Runes classes.

The ***Trader*** class is a concrete class that acts as an NPC (non-playable character) that is only responsible for buying and selling weapons to Players, following the SRP (Single Responsibility Principle) which states that a class should have only one responsibility. The Trader class extends the Actor abstract class which provides all the functionalities required by an Actor, reducing code repetition following DRY principle which is one of the benefits.

For the buying and selling of Items/Weapons, we won't be using a list as it is assumed that merchants do not have inventories to store the items. For each weapon item that can be sole or bought, we added a sellPrice and buyPrice attribute with type int. Then, we implemented

a **BuyWeaponAction** class and a **SellWeaponActionClass** which extend the Action abstract class. The purpose of these two classes is for the interactions between traders and players for making transactions. The BuyWeaponAction will have a dependency to the **Purchasable** interface, which represents Weapons that can be purchased from the trader (Uchigatana, Great Knife, Club), while the SellWeaponActionClass will have a dependency to the **Sellable** interface, which represents Weapons that can be sold to the trader (Uchigatana, Great Knife, Club, Grossmesser). These two subclasses, **BuyWeaponAction** and **SellWeaponActionClass** adhere to the SRP principle because they only have a single responsibility of buying or selling items.

Furthermore, we created a new ***Rune*** class which extends from the Item abstract class which will be representing the currency being used by players and traders in the game of Elden Ring. The **RuneManager** class will be responsible for storing the Player's rune value, as well as managing the value of the runes such as adding, subtracting, etc. This follows the SRP by solely having one responsibility, making the class to have a clear purpose which makes it easy to understand and maintain.

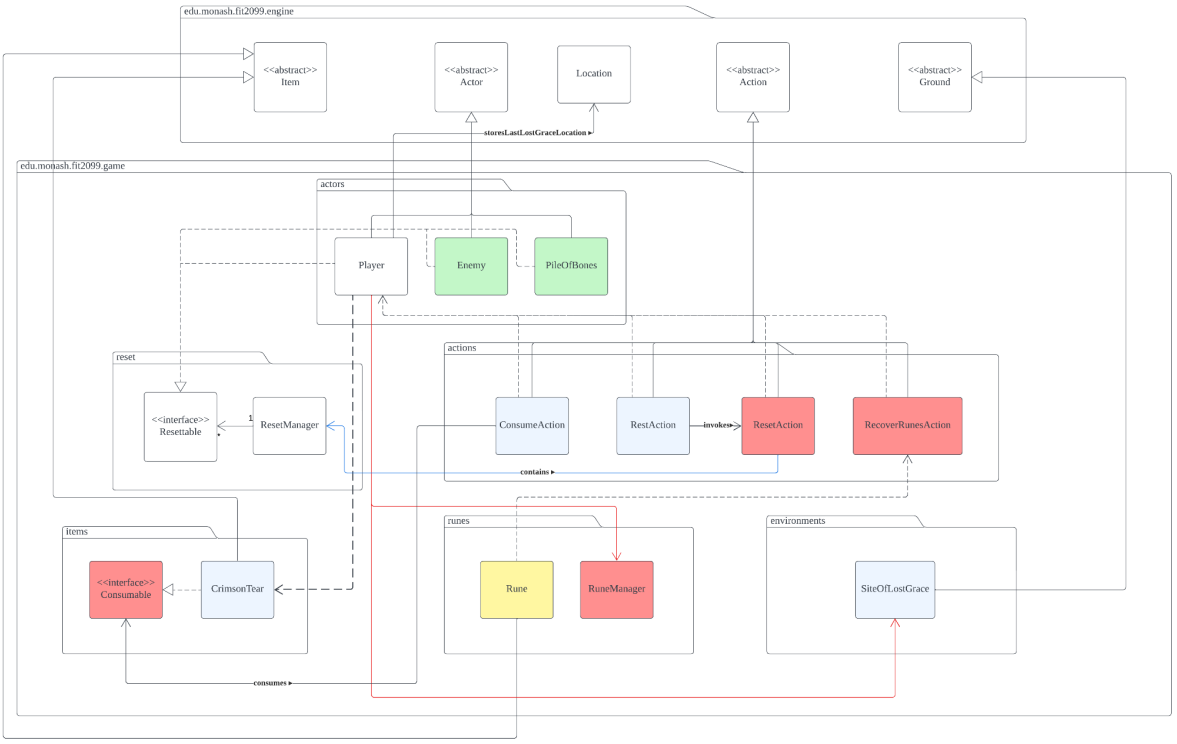
The Enemy abstract class is associated with the Rune class because enemies drop runes within a given range when killed. The Enemy class will have two extra constructor parameters, max_rune and min_rune of type int. The max_rune and min_rune is to find the random value of rune between the given ranges of the enemy that will be generated by the random value generator in the RandomGeneratorClass. Then, the action of adding runes to the player's total rune value when an enemy is killed is done in the DeathAction class's execute method. Even so, the Rune class is independent from the other classes that use Runes, following the DIP (Dependency Inversion Principle). This means that the implementation of Rune class will not be affected by the other classes that use it.

How this design can be extended in the future:

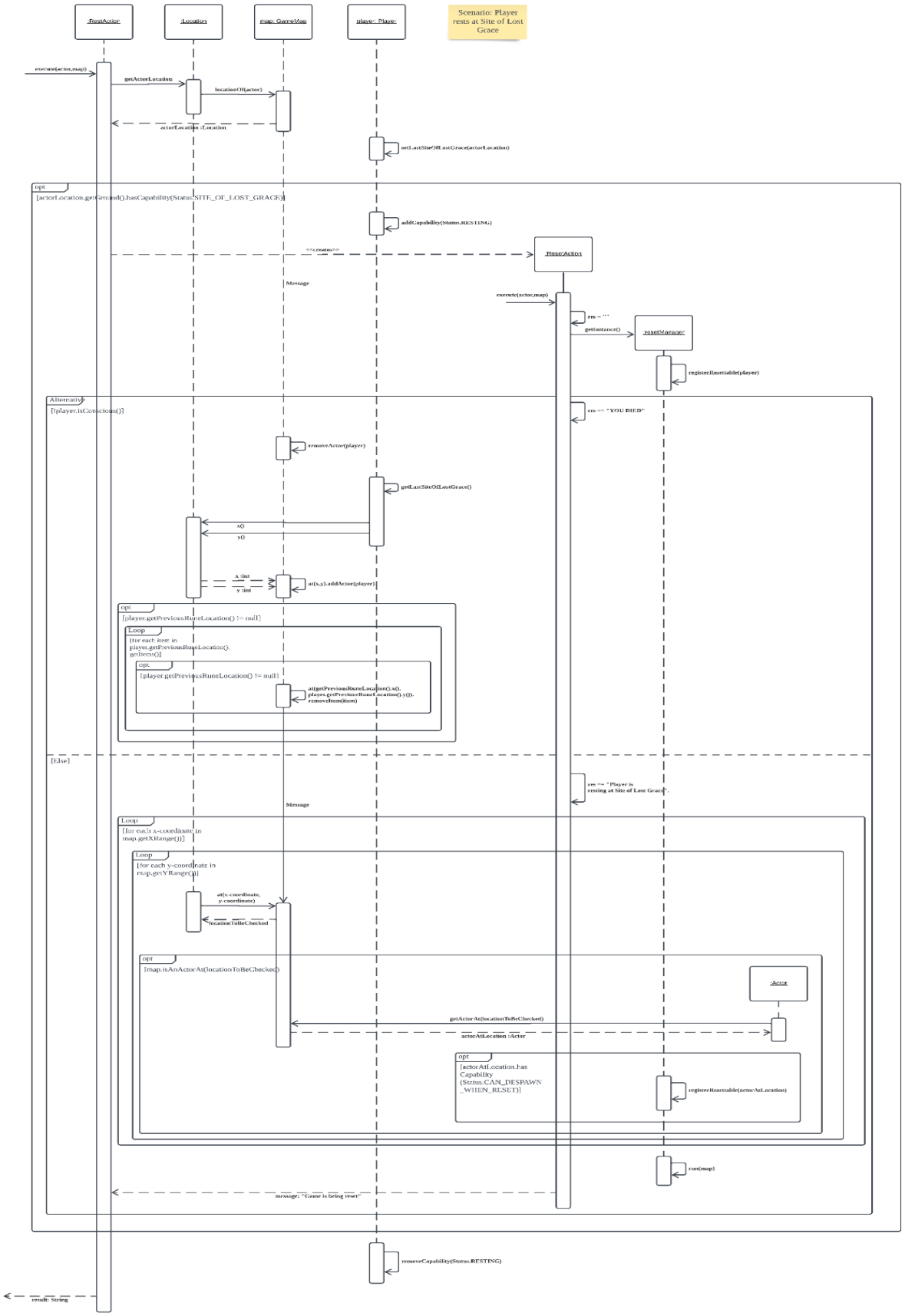
- New weapons that are added can implement the Purchasable interface if it can be bought, or the Sellable interface if it can be sold, or both. This will allow players to buy or sell weapons from/to the trader without having to modify the existing code. This follows the Open-closed principle, open for extension, closed for any modification.
- New enemies added will have their rune value auto-generated after giving the range value of the enemy in the constructor's superclass' parameter.

Requirement 3 - Grace & Game Reset

UML Diagram



Interaction Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the Grace and Game Reset classes.

The ***CrimsonTear*** class inherits from the Item class as it holds common attributes and methods with its own additional features. Composition is being used to show its dependency from Player to CrimsonTear class, because each player will obtain two CrimsonTear objects at the start of the game. CrimsonTear cannot be dropped, which restricts the behaviour of the original Item parent class which can be dropped. So this implementation will break the LSP principle which is a disadvantage for this class. This may be bad because it could lead to issues in the future if our code needs to be extended. In our implementation, CrimsonTear will not implement the Resettable interface, the number of uses left for the CrimsonTear will be kept count in the Player class. Therefore, the reset of the number of uses for the CrimsonTear will be done in the Player class instead.

The ***Consumable*** interface is for items that are consumable. CrimsonTears implements this interface because CrimsonTear can be consumed by the player. The implementation of the Consumable interface adheres to the ISP by ensuring that interfaces are segregated to only include the methods that are relevant to the implementing class. For example, the Consumable interface only includes methods that are relevant to consumable items.

The ***ConsumeAction*** class follows the principle of encapsulation by extending the abstract Action class and encapsulating the logic for consuming consumable items because we assume that not all items will be consumable. Again, composition is being used as Player and Item classes will be an attribute of the ConsumeAction class, which promotes flexibility and modularity. Both CrimsonTear class and ConsumeAction class follows the SRP because they both have a single responsibility, one is used to represent the logic of CrimsonTear's behaviour and one for the logic of consuming consumable items.

Subsequently, the ***SiteOfLostGrace*** class, which is a unique ground in the game, extends the abstract Ground class. Since only players can enter the LostGrace, we will be overriding the Ground class' canActorEnter method, where it returns true only if the actor is a Player instance. The LostGrace class follows the LSP principle, because it overrides the canActorEnter method with compatible behaviour which still sticks to the contract of the abstract class. Also, when the player dies, they will be respawned in the last sit of lost grace they have visited before, so there will be an association between the DeathAction class and the GameMap class, as well as Player class to LostGrace class to store the last visited LostGrace ground.

We will also be creating a ***RestAction, ResetAction*** class which extends the Action abstract class. RestAction can only be invoked when the player is in the ***SiteOfLostGrace*** ground. The ResetAction class will have an association to the ***ResetManager*** class, which manages the list of resettable objects when the game is reset. When players are resting at the SiteOfLostGrace, the ResetAction will be invoked, causing the following to change:

1) HP of player will be reset to maximum hit points
2) The number of uses for CrimsonTear will be reset to the maximum number of uses

3) Enemies that can be despawned will despawn from the map

When the player dies, the game will not end. Instead, the game will be reset and the player will be respawned at the last ***SiteOfLostGrace*** visited. Therefore, the player will have an association to Location to keep track of the location of the last SiteOfLostGrace visited (take note that it is not the SiteOfLostGrace object stored but the location). During the reset of the game, the statements stated above (1-3) will also be invoked. Also when the player dies, the runes will be dropped on the location just before the player dies. To do this, we will be storing a list of player location history in the Player class as an attribute. So when the player dies, the rune will be dropped at the second last location which is the location before they died.

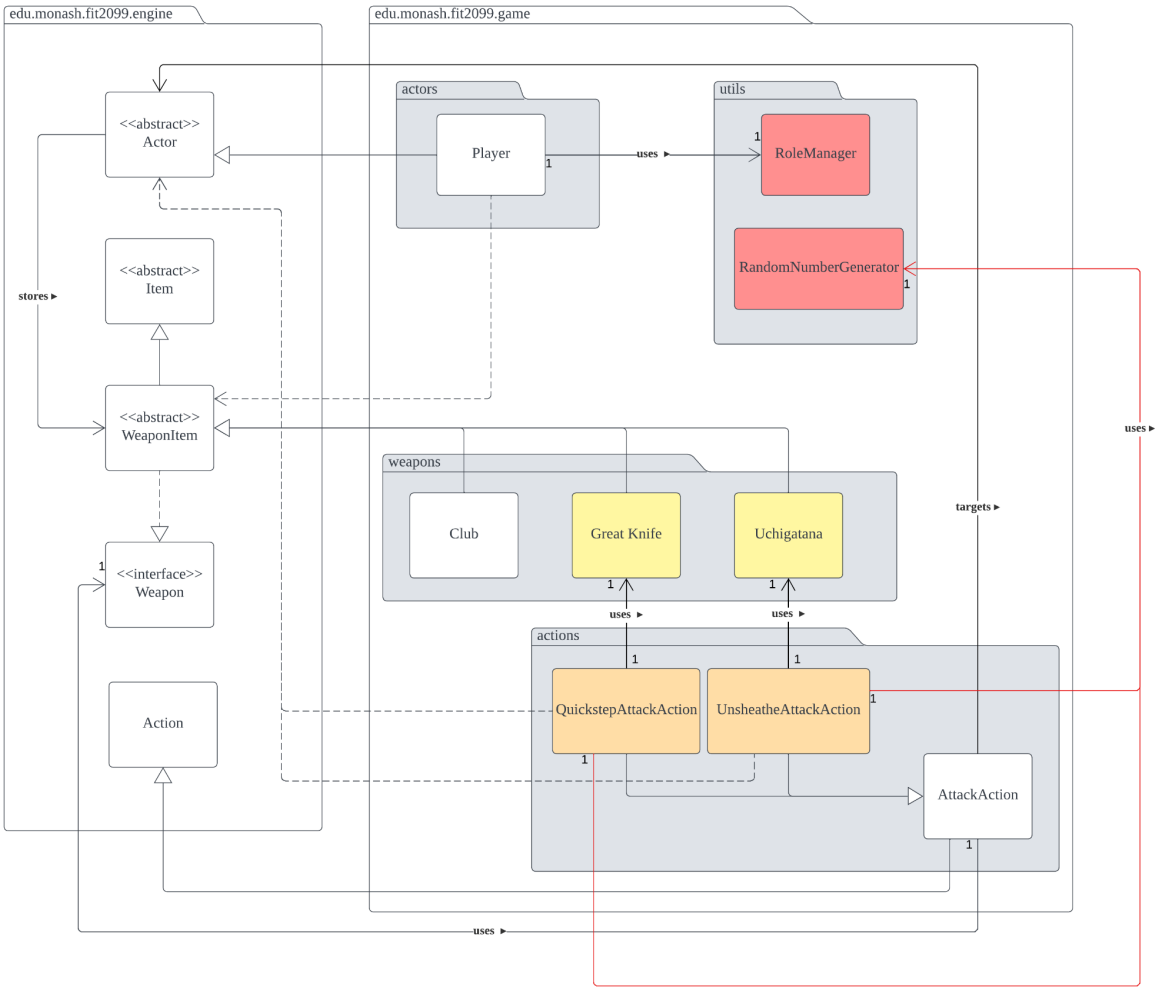
The two classes, Player, Enemy classes will have specified changes when the game is being reset, so they will be implementing the Resettable interface and have the rest() method being overridden to state their logic or behaviour when the game is being reset. This design follows the SRP principle as each of these classes will be responsible for resetting its own state, and this will allow extensibility and maintainability for further game versions in case other classes are resettable. The design allows for extensibility in the future by implementing the Resettable interface and providing the rest() method, which can allow for other classes to be resettable and easily maintainable.

How this design can be extended in the future:

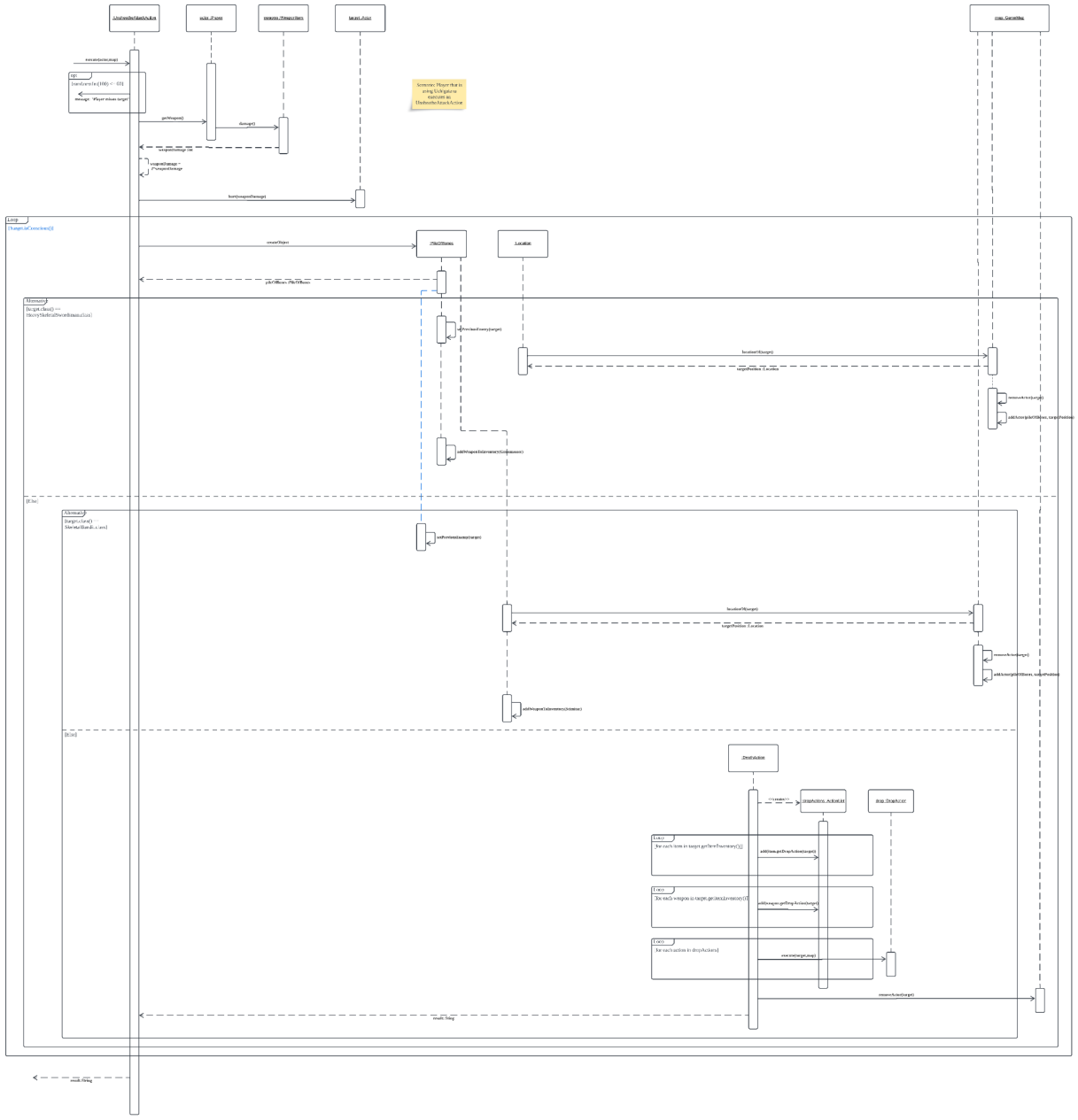
- A new type of ground can be introduced that affects the movement of the player or enemies. To implement this, a new class can be created that inherits from the Ground class, and the canActorEnter method can be overridden to reflect the new behaviour.

Requirement 4 - Classes (Combat & Archetypes)

UML Diagram



Interaction Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the combat and archetypes of players.

At the start of the game, the players will have to choose between the player classes they want to play, which are Samurai, Bandit and Wretch for the current version. The three classes differ from each other because they will have different hit points and they will also be given different starting weapons at the start of the game. Instead of using Inheritance to represent the different player classes, we will be using the RoleManager class to manage

the player classes at the start of the game. In the player class, another method is created to display a menu with the three player classes to choose from at the start of the game. Based on the choices of the player, different weapons will be added to the player's inventory as a starting weapon. For example, ***Samurai*** will be starting with ***Uchigatana***, ***Bandit*** with ***Great Knife*** and ***Wretch*** with ***Club***. Through this design, we will be able to avoid Multiple Inheritance by creating Samurai, Wretch and Bandit classes which are exactly the same as players, which can become too complex and prone to getting a lot of errors. The cons of this design is that it does not address the possibility of adding new player classes in the future. If it does, then the RoleManger class and the Player class's constructor will have to be modified, breaking the OCP principle.

All three starting weapons, ***Uchigatana, Great Knife*** and ***Club*** extend the WeaponItem class which extends the Item abstract class and implements the Weapon interface. This allows the creation of new Weapon types without having to modify the existing code. By extending the WeaponItem class, the three classes, ***Uchigatana***, ***Great Knife*** and ***Club***, the common attributes and methods that are implemented in the WeaponItem abstract class as well as the Item abstract class can be reused. This follows the DRY (Don't Repeat Yourself) principle. Any new additional attributes and behaviours can be implemented in the classes.

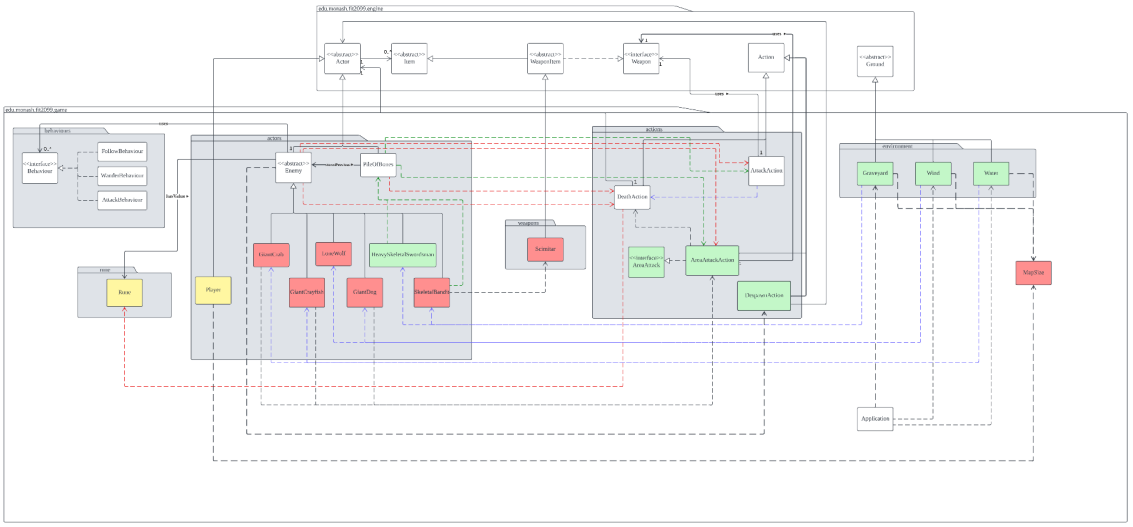
There are two unique skills, ***Unsheathe*** skill for ***Uchigatana*** users to deal 2x damage of the weapon with a hitRate of 60% to attack the enemy and ***Quickstep*** skill for ***Great Knife*** users that deals normal damage to the weapon to the enemy, and after attacking, the user moves away from the enemy to evade their attack. These two unique skills are created as two separate subclasses of the AttackAction class, ***UnsheatheAttackAction*** and ***QuickstepAttackAction***. The execute method will be overridden to implement the behaviours of the unique skills which follows Polymorphism. Here, not only Inheritance is being used, but also OCP (Open-Closed Principle) as the AttackAtion class is open for extension and closed for any modification to the code, meaning that the behaviour of the class can be extended through the creation of new concrete subclasses without modifying the existing code. Furthermore, AttackAction is kept as non-abstract because the users that use Uchigatana or Great Knife weapons can still use the parent AttackAction class for a basic attack, and also their respective unique skills that are subclasses of the AttackAction class.

How this design can be extended in the future:

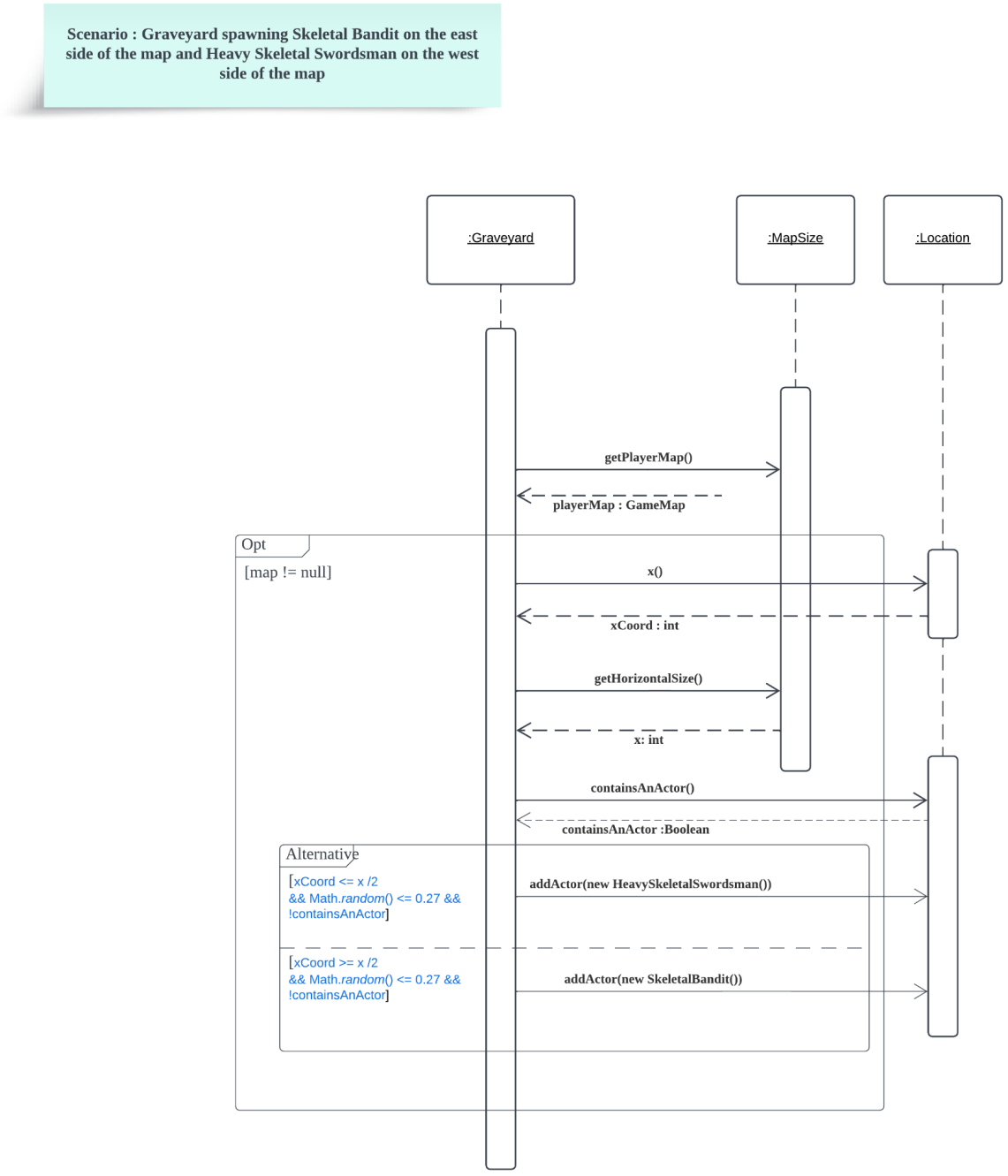
- To add new player classes, The RoleManager class can be modified to allow for the creation of new player classes in the future. This can be done by adding new methods to the RoleManager class to handle the creation and management of new player classes.
- To add new skills, they can be added to the game system by creating new subclasses of the AttackAction class. The new subclasses can override the execute method to implement the behaviours of the new skill. This follows the Open-Closed Principle, as the AttackAction class is open for extension and closed for modification.

Requirement 5 - More Enemies (HD Requirement)

UML Diagram



Interaction Diagram



Design Rationale

From our design in requirement 1, we are able to create new enemies as we have designed it following the DRY principle by creating an abstract Enemy class. Hence, 3 new concrete classes are extended from the **Enemy class** which are the **Skeletal Bandit**, **GiantDog** and **GiantCrayfish**. The implementation for the 3 new enemies are similar to the enemies that we have created in requirement 1. Therefore, no new classes are required to be created other than the Scimitar class which is the weapon used by the Skeletal Bandit. In addition,

the DeathAction will have a dependency with the Rune class as the enemies are able to drop runes within a specific range if it is killed. This shows the benefit of using the Enemy abstract class which allows new enemies to be added to the game and have all of the characteristics an enemy should have.

The SkeletalBandit class should be implemented similarly to the HeavySkeletalSwordsman as they are the same type. The only difference is that instead of the Grossmesser weapon, the skeletal bandit will have a Scimitar. **Scimitar** class is extended from the WeaponItem abstract class which extends from the Item abstract class and implements the Weapon interface. This implementation allows the Scimitar to have the ability as any weapon including the ability to be bought from and sold to the Trader class which is similarly implemented in requirement 2. This shows that we are following the DRY principle which is one of the pros.

Although the GiantDog and LoneWolf are the same type, the implementation of the GiantDog would be slightly different as it has slam attack (single target) as an intrinsic weapon instead of bite and the GiantDog will have a dependency to the **AreaAttackAction** that implements **AreaAttack** interface which allows the GiantDog to cause damage to all creatures in its surroundings as a special skill. This is similar to the implementation of GiantCrayfish which has the same type as GiantCrab.

Lastly, instead of creating an interface or an abstract class to implement the east spawn or west spawn of **Graveyard**, **Wind** and **Water** classes, We have created a MapSize class which has a static method that allows us to obtain the size of the map that the player is currently in. This shows that there is a dependency of the player class towards the MapSize class to set the size of the map. By using the static method, we are able to get the size of the map and implement the spawning of the enemies based on the location of the**Graveyard**, **Wind** and **Water** classes. The benefit of using a static method is that we are able to invoke the static method without creating an instance of the class which is efficient and helps us to reduce code repetition. However, it may lead to excessive use which will result in code smell. Another disadvantage is that we have similar code to check if the location is in the east or west in each of the environments. This causes us to follow the WET principle.

How this design can be extended in the future:

- If a new environment would like to have the characteristics of spawning different actors based on their position. The new environment is able to use the static method to get the size of the map and modify the environment class accordingly.