`

MONASH
University

FIT2099
Assessment 2
UML, Sequence Diagrams, Design Rationale

Team: LAB7_GROUP8

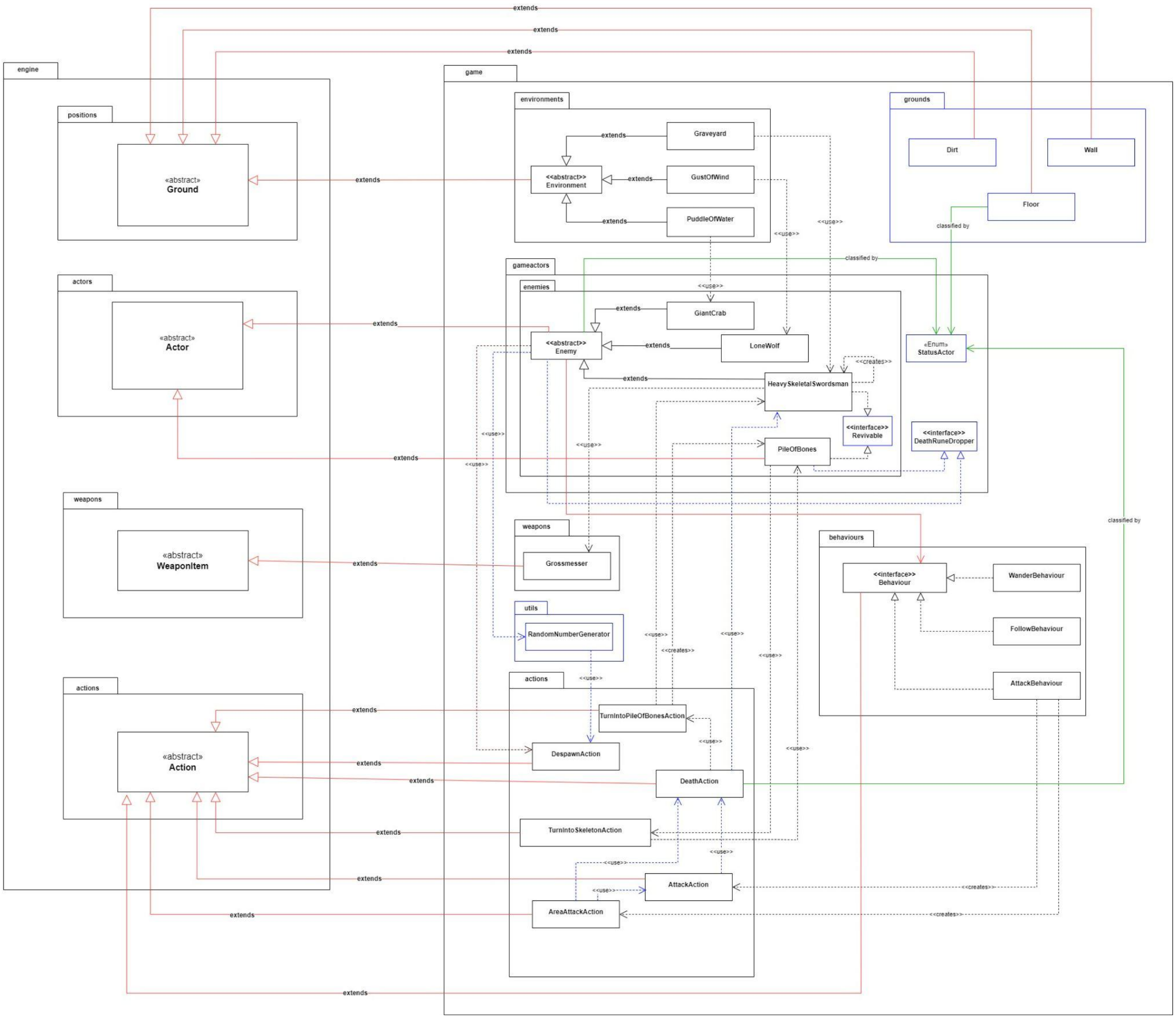
Team members:

1. **Tanul Gupta**
2. **Satoshi Kashima**
3. **Aditti Gupta**

Date: 03/05/2023

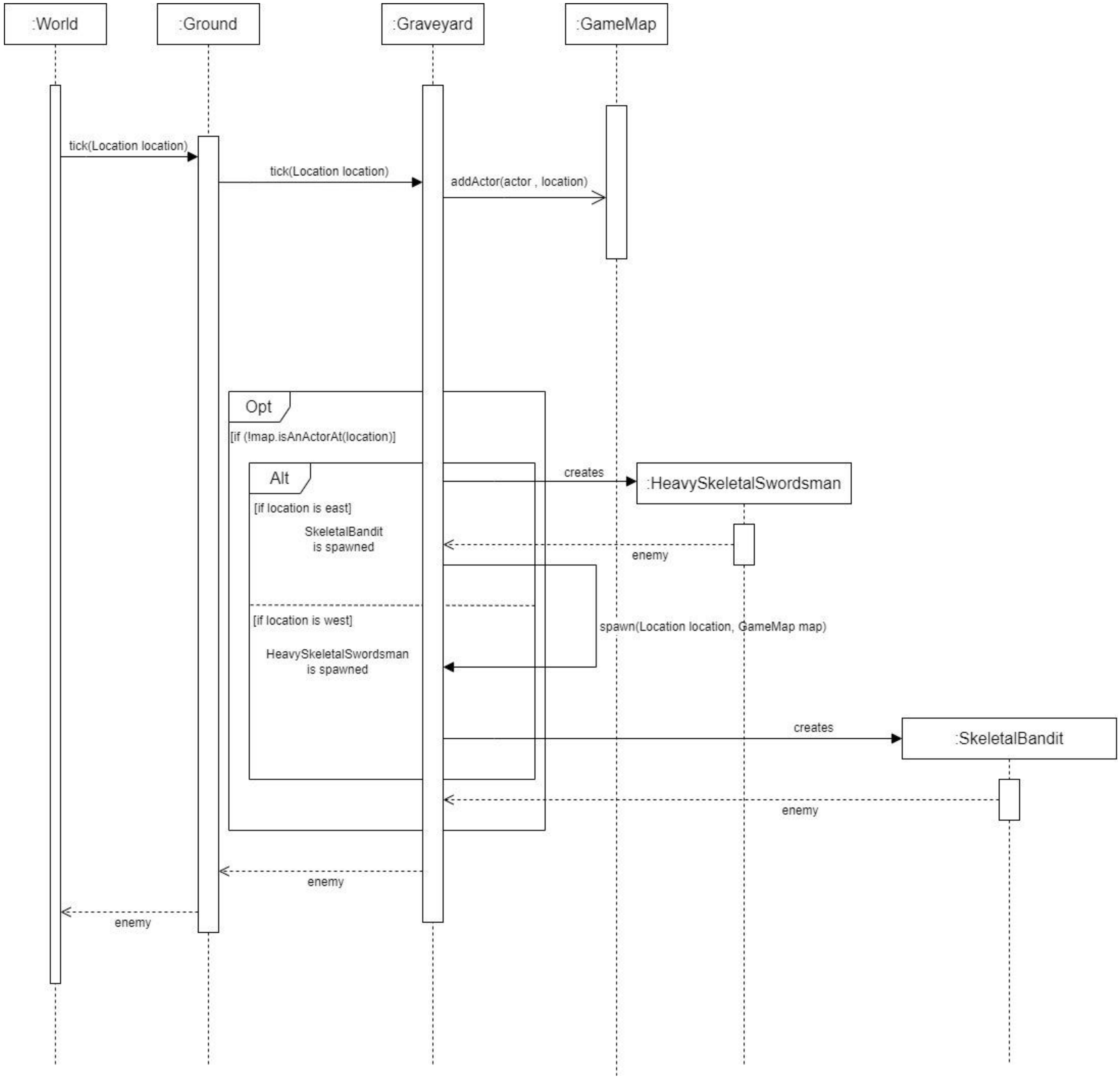
REQ1:

UML



(The classes that are different from A1 are highlighted in blue)

Sequence diagram - REQ1 - Spawning



Design rationale:

Environments:

Environment (abstract class):

In the package named environments, an abstract class called Environment that extends Ground from engine is made. It contains a constructor and an abstract method for spawning an Enemy on a GameMap. It also has a method to determine if a given Location is in the eastern half of the GameMap, to differentiate spawning for different enemies with their respective locations, i.e, East or West (specified below). The Environment class is designed to be a base class for various environments in the game (Graveyard, GustOfWind and PuddleOfWater), where enemies can spawn.

Pros:

SOLID Principles Used:

- Single Responsibility Principle (SRP): The Environment class has a single responsibility of providing a base class for environments in the game, with the capability to spawn enemies. The tick method is responsible for checking for the presence of enemies and spawning them if none are found. The spawn method is responsible for actually creating an enemy in the given Location and GameMap.
- Open-Closed Principle (OCP): The Environment class is designed to be open for extension and closed for modification. The class is designed to be extended by subclasses to implement the actual spawning logic. The Environment class itself does not need to be modified for this.
- Liskov Substitution Principle (LSP): Since Environment is an abstract class, it cannot be instantiated, but it can be used as a type for its subclasses. This means that wherever an Environment object is expected, any of its subclasses can be used without affecting the correctness of the program.
- Dependency Inversion Principle (DIP): The Environment class depends on the GameMap, Location, Ground, Display, and RandomNumberGenerator classes/interfaces to function. However, it does not create any of these dependencies directly. Instead, they are passed as parameters to the spawn and tick methods. This allows for greater flexibility in choosing the implementation of these dependencies at runtime.

Cons and improvements:

- S - Single Responsibility Principle: The Environment class should have only one responsibility. The current implementation has the responsibility of spawning an Enemy as well as determining if a location is in the eastern half of the GameMap. To adhere to the SRP, we should split these responsibilities into two separate classes.
- O - Open-Closed Principle: The Environment class should be open for extension but closed for modification. Currently, the spawn method is abstract, which allows for subclasses to extend the functionality, but the tick method is not open for extension. We can modify the tick method to be open for extension by refactoring it to call a separate method that subclasses can override.

The Graveyard (HeavySkeleonlSwordsman), GustOfWind (LoneWolf), and PuddleOfWater(GiantCrab) classes are subclasses of the Environment abstract class in a game. They represent specific types of game environments where enemies can spawn. Each class overrides the spawn method to randomly generate and return an instance of one or two types of enemy objects.

Gameactors:

Enemies:

Enemy (abstract class):

This is an abstract class Enemy that extends Actor, which represents an enemy in a game. It has a constructor that takes the name, display character, and hit points of the enemy, and sets its capabilities as an enemy. It also contains a list of behaviours with their priorities, an intrinsic weapon, and methods to determine the allowable actions for the enemy and if it can target other actors.

Pros:

SOLID Principles Used:

- Single Responsibility Principle: The Enemy class has a single responsibility of representing an enemy in a game and its associated behaviors.
- Open-Closed Principle: The Enemy class is designed to be extended for different types of enemies and their behaviors, making it open for extension but closed for modification.
- Liskov Substitution Principle: The Enemy class is a subtype of the Actor class and can be used wherever an Actor is expected.
- Interface Segregation Principle: The Enemy class only exposes methods that are necessary for its behaviors and interactions with other game objects.
- Dependency Inversion Principle: The Enemy class depends on abstractions such as the Behaviour interface and the Resettable interface, making it easier to modify and extend.

Cons and improvements:

- The allowableActions() method should take the map and player as arguments, rather than assuming that the player is a global variable accessible by the Enemy class.
- The intrinsicWeapon and behaviours Map should be declared as private and have public getter and setter methods to access and modify them.

HeavySkeletonSwordsman:

The HeavySkeletonSwordsman class is a subclass of the Enemy class. It turns into a PileOfBones when it dies, can be be revived back. It overrides the revive method from the parent class to return a new instance of itself, and it adds a Grossmesser weapon to its inventory in the constructor. Revivable interface is implemented here.

PileOfBones:
The PileOfBones class is a simple game object that can be revived back into a Skeleton after 3 turns. The class extends the Actor class and implements the Resettable and DeathRuneDroppper interfaces. The class contains a private boolean checkForRevive() method that checks whether the PileOfBones has been alive for 3 turns and can be revived back into a Skeleton. The playTurn() method overrides the same method in the Actor class to allow the PileOfBones to revive back into a Skeleton after 3 turns. The getDeathRune() method returns the Rune that should be dropped by the PileOfBones when it dies. Revivable interface is implemented here.

LoneWolf:
The LoneWolf class is a subclass of Enemy and represents a type of enemy in a game. The LoneWolf has an intrinsic weapon bite which allows it to deal damage to its enemies.

GiantCrab:
A class named GiantCrab is made that extends the OceanEnemy class that can perform a slam attack. The GiantCrab constructor calls the constructor of its parent class and sets the name, display character, hit points, and maximum damage of the crab.

The getIntrinsicWeapon method of GiantCrab returns an instance of IntrinsicWeapon class, which is a weapon that is built into the creature's anatomy and does not require the creature to wield a separate object.

DeathRuneDropper: (Interface)
The DeathRuneDropper interface is designed to represent an enemy that drops a death rune when defeated in a game. This interface defines a single method getDeathRune() that returns the death rune dropped by the enemy. By implementing this interface, the enemies can drop death runes when they are defeated in the game.

Revivable: (Interface)
The purpose of this is to provide an interface for enemy actors in a game that can be revived. The Revivable interface has a single method named revive() which returns a new instance of the revived enemy.

StatusActor: (Enumerator)
An enumeration is used to represent various statuses that game actors can have. This approach provides a convenient way to manage the different statuses that game actors may have, and allows for easy comparisons and switches based on the actor's status. This basically classifies the gameactors so allow/prevent certain actions unique to each of them.

Weapons:

Grossmesser:

The class Grossmesser is a subclass of the WeaponItem class. The Grossmesser is a simple weapon that can be used to attack an enemy. The Grossmesser implements the Sellable interface, which defines a method to get the selling price of the weapon. It also has a capability for AreaAttckAction when spinning attack is prompted.

Pros:

SOLID Principles Used:

- Single Responsibility Principle (SRP): The "Grossmesser" class has a single responsibility, which is to represent a simple weapon that can be used to attack an enemy. The class has a constructor that initializes the weapon's attributes and a method to get the selling price of the weapon.
- Open-Closed Principle (OCP): The "Grossmesser" class is open for extension but closed for modification. The class can be extended to create new types of weapons, but the existing implementation of the "Grossmesser" class is not modified.
- Liskov Substitution Principle (LSP): The "Grossmesser" class is a subclass of the "WeaponItem" class and can be used wherever a "WeaponItem" object is expected.
- Interface Segregation Principle (ISP): The "Sellable" interface has only one method, "getSellingPrice," which is implemented by the "Grossmesser" class. This principle is followed by having a small and cohesive interface.
- Dependency Inversion Principle (DIP): The "Grossmesser" class has no direct dependencies on any other class or module, following the Dependency Inversion Principle.

Cons and improvements:

- The code could use more comments to explain the purpose of the methods and attributes.
- The constructor's parameters could be defined as constants or variables to improve code readability.

Grounds:

Dirt:

The Dirt class represents a type of Ground in a game world, which is essentially just bare dirt with no special properties or behavior. The class extends the Ground class. No gameactor has any restrictions to access this ground.

Floor:

The Floor class extends ground. It uses the canActorEnter method to restrict access to the Floor object based on the capabilities of the actors trying to enter, i.e., the player and the trader only.
No enemy is allowed to enter the floor ground.

Wall:

The Wall class extends the Ground class. It represents a type of ground object that blocks actors from entering and thrown objects from passing through it. The canActorEnter() method always returns false, indicating that actors cannot enter this ground type. The blocksThrownObjects() method always returns true, indicating that thrown objects cannot pass through the wall.

Behaviours:

Behaviour: (Interface)

Behaviour is an interface which defines a factory method getAction that returns an Action object, representing a kind of objective that an Actor can have. The Behaviour interface provides a way to modularize the code that decides what to do and allows it to be reused in other Actors via delegation instead of inheritance.

Pros:

SOLID Principles Used:

- Single Responsibility Principle (SRP): The Behaviour interface has a single responsibility - to define a factory method that returns an Action object representing an Actor's objective. The implementation of that objective is delegated to other classes.
- Open/Closed Principle (OCP): The Behaviour interface is open for extension but closed for modification. It defines a factory method that can be implemented by any concrete Behaviour class, without modifying the existing code.
- Liskov Substitution Principle (LSP): The Behaviour interface can be substituted with any of its concrete implementation classes, without affecting the correctness of the program.
- Interface Segregation Principle (ISP): The Behaviour interface only contains the necessary methods for creating actions, and nothing more. There are no unnecessary methods in the interface.
- Dependency Inversion Principle (DIP): The Behaviour interface defines an abstraction for creating actions, which can be used by any Actor without knowing the implementation details of the concrete Behaviour classes.

Cons and improvements:

- The code could benefit from adding more methods to the interface to provide more flexibility in creating behaviours.
- The Behaviour interface could be renamed to something more descriptive

AttackBehaviour:

The AttackBehaviour class is designed to provide an attack action for an enemy towards the player or other enemies. The class implements the Behaviour interface, which ensures that the getAction() method is implemented. The class takes the player as an argument in the constructor because it needs to check if the player has moved in the same turn or not.

The AttackBehaviour class first checks if the player has moved in the same turn or not. If the player has moved, it returns null, indicating that no action is possible. Then it checks if the enemy has a weapon or not. If it has a weapon, it determines if the weapon has the capability for targeted or area attacks.

If the weapon has an area attack capability, it returns an AreaAttackAction object. If the weapon has a targeted attack capability, it returns an AttackAction object. If the enemy doesn't have any weapon, it performs an intrinsic weapon attack, and if it has a weapon, it performs a regular weapon attack.
The class then collects all possible enemies around the current location of the enemy and returns the appropriate action based on the attack capability of the weapon.

FollowBehaviour:
The FollowBehaviour class is responsible for providing the logic that allows an actor to move closer to a target actor by one step. It computes the Manhattan distance between the current location of the actor and the target actor, and then checks the exits in the current location to see if any of them lead to a location that is closer to the target actor. If it finds a closer location, it creates a MoveActorAction object that moves the actor to the new location.

WanderBehaviour:
WanderBehaviour class allows an actor to randomly wander around the game map. It checks for all possible exits from the current location of the actor, and if the actor can enter the destination location, it adds the corresponding MoveAction to a list of possible actions. If at least one action is available, the getAction method randomly selects one action from the list and returns it. Otherwise, it returns null.

Actions:

AttackAction:

The AttackAction class represents an action for an actor to attack another actor with a weapon. The class contains the target actor that will be attacked, the direction of the attack for display purposes, the weapon used for the attack, and a random number generator for calculating the chance to hit the target. The class has two constructors, one for passing in a specific weapon and another for using the intrinsic weapon of the actor performing the attack.

The execute() method of the AttackAction class first checks if the target actor has the CANNOT_BE_ATTACKED capability. If the target actor has this capability, the attack will not be executed. Then, the chance to hit the target is calculated based on the chance to hit of the weapon used. If the attack misses, a message is returned indicating the miss. If the attack hits, damage is dealt to the target based on the damage of the weapon used. If the target is not conscious after the attack, a DeathAction or a TurnIntoPileOfBonesAction is executed based on the target's EnemyType capability.

Pros:

SOLID Principles Used:

- Single Responsibility Principle (SRP): The AttackAction class has a single responsibility, which is to execute an attack action. The class only has methods and attributes that are relevant to this responsibility.
- Open-Closed Principle (OCP): The AttackAction class is open for extension through its constructors, which allow the user to pass in a specific weapon or use the intrinsic weapon of the actor performing the attack. The class is closed for modification, as there are no public methods that modify the class.
- Liskov Substitution Principle (LSP): The AttackAction class does not have any subclasses, so LSP does not apply.
- Interface Segregation Principle (ISP): The AttackAction class does not implement any interfaces, so ISP does not apply.
- Dependency Inversion Principle (DIP): The AttackAction class depends on other classes in the game.actions, game.gameactors, and edu.monash.fit2099.engine packages, but does not depend on any concrete implementations.

Cons and improvements:

- Separation of Concerns: The AttackAction class could be further improved by separating the concern of calculating the chance to hit the target into a separate class, as this is a single responsibility on its own.
- Dependency Injection: The Random class dependency could be injected into the AttackAction class through its constructor, which would make it easier to test the class.

AreaAttackAction:

The AreaAttackAction class represents an action where an actor attacks all the other actors in the surrounding area of the game map. This action can be done either with a specified Weapon or with the actor's intrinsic weapon.

DespawnAction:

The DespawnAction class represents the action of removing an Actor from a GameMap. The execute method takes an Actor and a GameMap as parameters and removes the Actor from the GameMap.

DeathAction:

This class extends the action class and is executed when an actor is killed in the game. It drops items and weapons carried by the killed actor to the location in the game map where the actor was killed. If the killed actor is a player, the player will drop a Rune, which can be picked up by the player. If the killed actor is an enemy, the enemy will drop all of its items and weapons, and a Rune will be transferred to the player who killed it. The class also handles respawning the player if the killed actor was a player. (for later parts)

TurnIntoPileOfBonesAction:
The code represents an action that can be performed by a Skeleton actor in a game. This action allows the Skeleton actor to turn into a PileOfBones actor. The execute method first removes the Skeleton actor from its current location on the map, and then creates a new instance of PileOfBones actor at the same location where the Skeleton actor was previously located.

TurnIntoPileOfSkeletonAction:
TurnIntoPileOfSkeletonAction class, implements the action of turning a PileOfBones object into a Skeleton object and reviving it. The action is executed by calling the execute() method. The method then revives the target Skeleton object and adds it to the same location as the original PileOfBones object in the GameMap.

Utils:

RandomNumberGenerator:
The RandomNumberGenerator class provides static methods to generate random numbers. These methods use the Java built-in class Random to generate random numbers, that are further used in the code when needed.

Pros:

SOLID Principles Used:

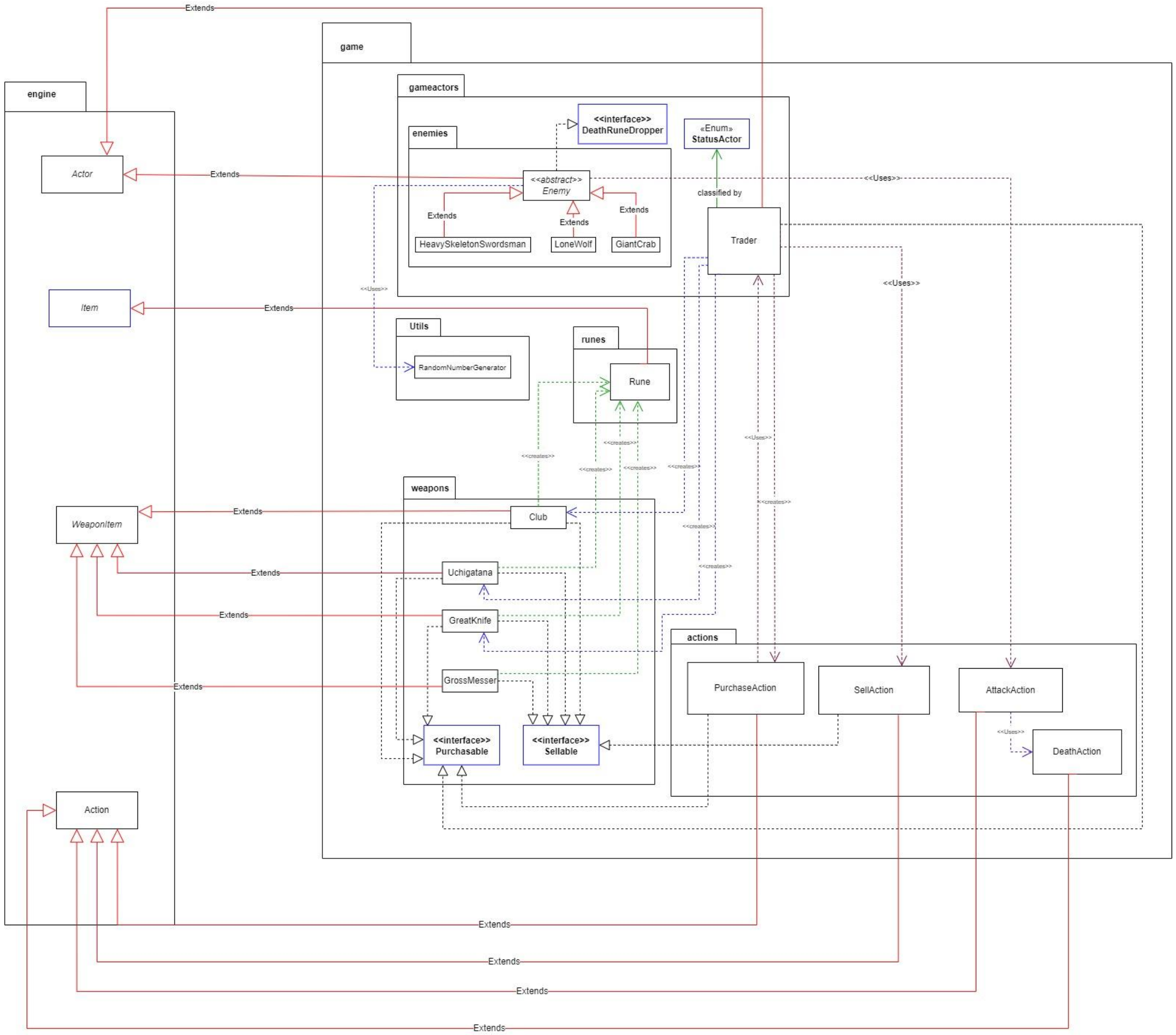
- This code follows the Single Responsibility Principle (SRP) because it only provides methods for generating random numbers and does not perform any other unrelated tasks.
- This code also follows the Open/Closed Principle (OCP) because it is easy to add more functionality to this class without modifying the existing code. For example, one could add a new method to generate random double numbers without modifying any of the existing methods.

Cons and improvements:

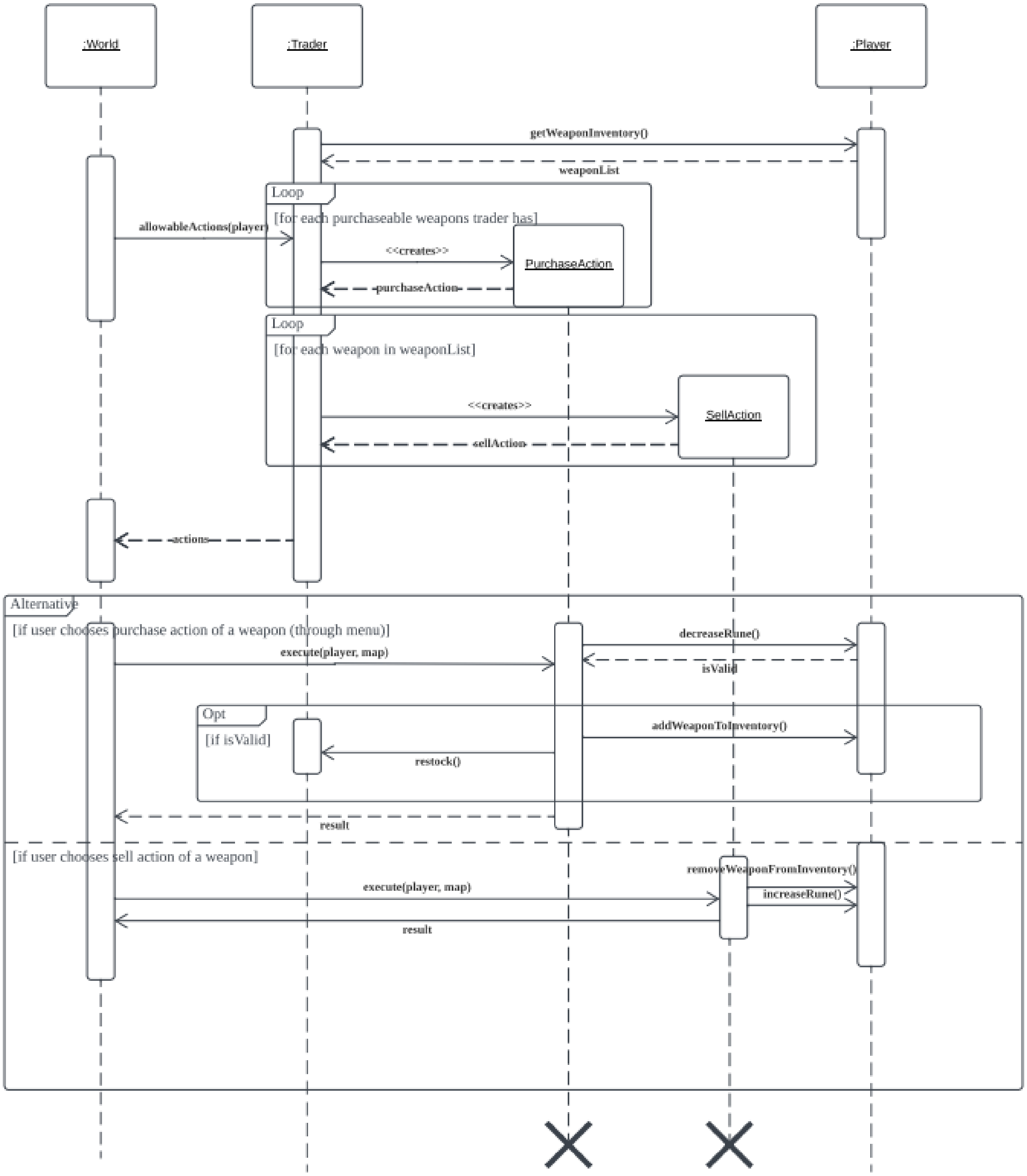
- Using a single instance of the Random class is recommended to generate random numbers, as creating a new instance for each method call is not efficient. Therefore, it would be better to have a private static Random instance variable and use it in all the methods to generate random numbers.

REQ 2

UML



Sequence Diagram (for purchase/sell action)



not explicitly delected,
but will no longer be
referenced by variables

Design Goals:

1. Rune: the player is granted runes, the in-game currency when they kill any of the three enemies (Heavy Skeletal Swordsman, Lone Wolf or Giant Crab).
2. Trader: the trader whom the player can purchase from and sell to
3. The Runes dropped will be random in a range specific to each of the enemies specified above.

Design changes from Assignment1

1. We created an interface DeathRuneDropper which is implemented by actors who drop runes when being killed.
    a. Some actors require dropping runes when it's dead. To follow OCP and DRY, we created this interface. In the future, if we get more complex actors, with or without dropping runes, we can make use of this interface without modification of existing code.
    b. The third design goal is achieved through implementation of this interface in enemy class. It uses random number generator and attributes representing minimum and maximum amount of runes dropped.
2. We created Purchaseable and Sellable interface
    a. This enables consistent access to the selling/purchasing price of each weapon. If a weapon cannot be sold/purchased, it does not implement these classes. It also enforces the price of each weapon to be stored in its corresponding class, avoiding usage of literal integers everytime this weapon needs to be sold/purchased.It is more object oriented. It follows OCP and ISP because in the future if we have an item/weapon or even an actor that can be sold/purhcased, they can just implement these interfaces.
    b. Note, however, that we need to downcast from WeaponItem to Purchaseable/Sellable when actually using the implemented methods.

Implementation details - Trader:

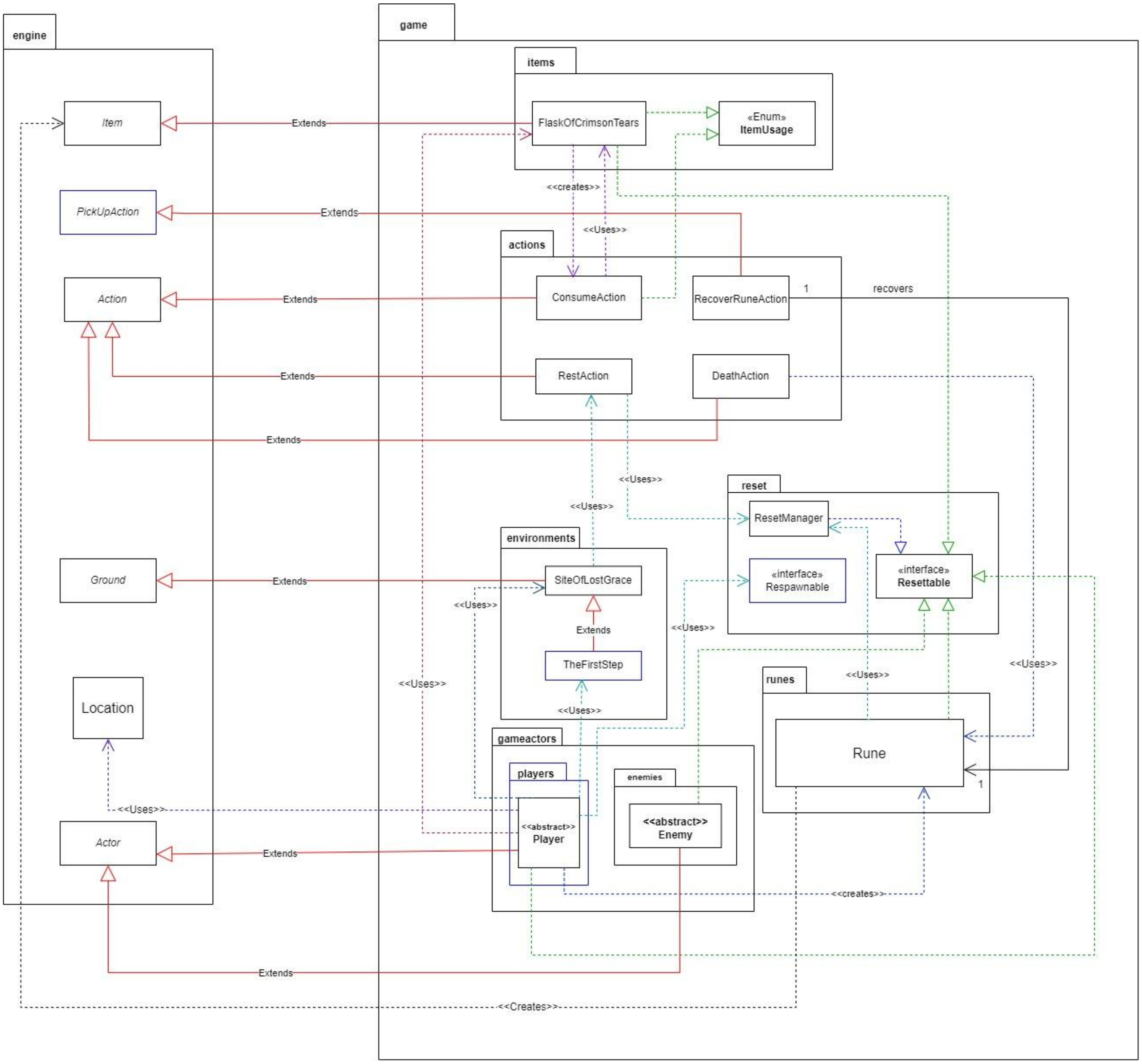
- Trader extends the "Actor" class and has a list of sellable weapons, which are instances of the "WeaponItem" class.
- The "Trader" class has a constructor that initializes the "sellableWeaponItems" list with some default weapons. It also adds two capabilities to the NPC using the "addCapability" method inherited from the "Actor" class. These capabilities indicate that the NPC is a trader and cannot be attacked.
- The "Trader" class has three methods:
    - "playTurn": This method overrides the "playTurn" method inherited from the "Actor" class. It doesn't perform any action and returns a "DoNothingAction" object, indicating that the NPC doesn't actively do anything during the game turn.
    - "restock": This method takes a "WeaponItem" object as an argument and removes it from the "sellableWeaponItems" list. It then calls the "restock" method on the "Purchasable" interface implemented by the "WeaponItem" object to add a new instance of the weapon to the list. We need this method because once a weapon is sold, the same weapon object should not be sold again by the trader.
    - "allowableActions": This method overrides the one in Actor class. It returns an "ActionList" object that contains two types of actions: "SellAction" objects for each weapon in the player's inventory and "PurchaseAction" objects for each weapon in the "sellableWeaponItems" list.

Implementation details - Rune

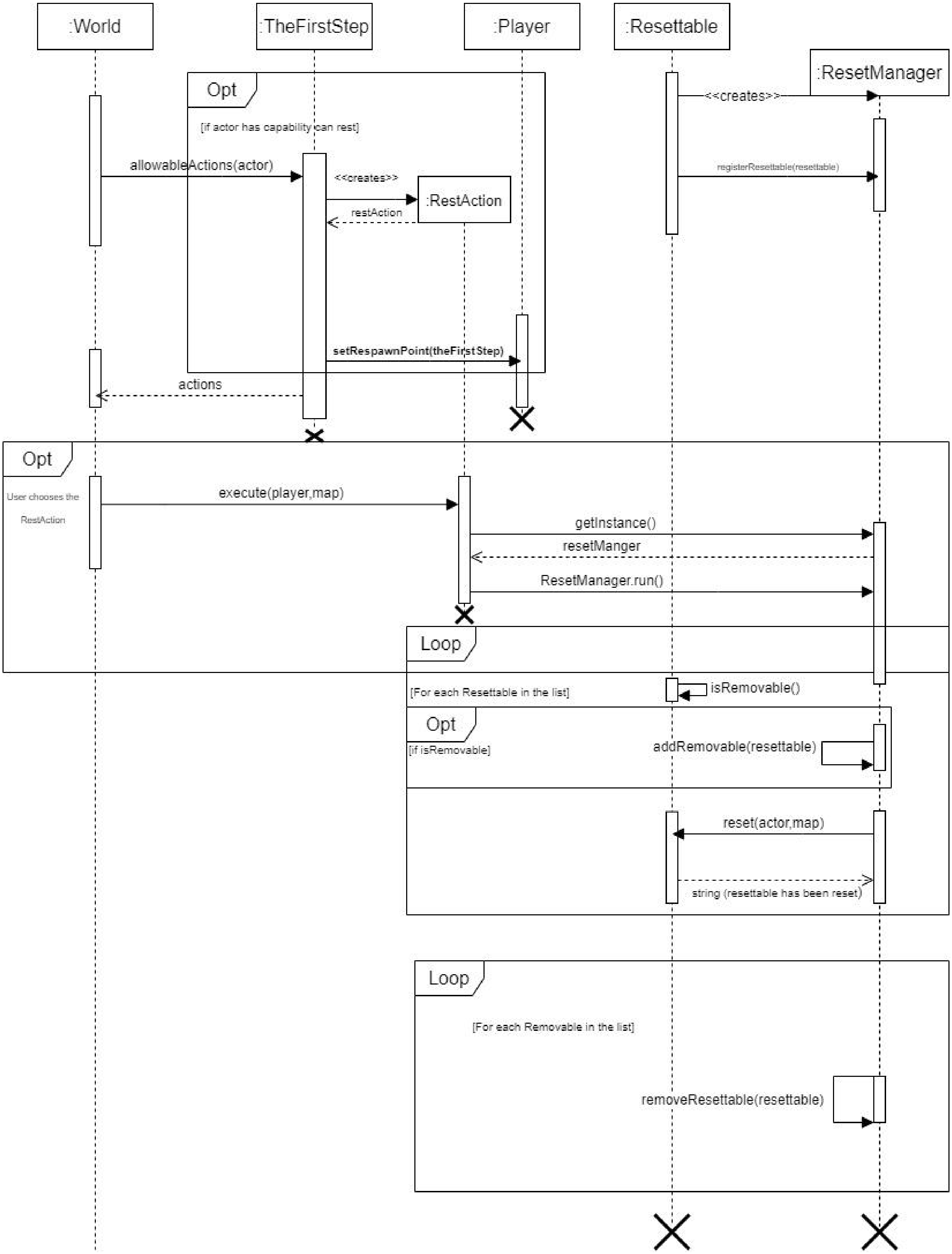
- It can be constructed with an initial amount of currency, or with zero currency.
- It has some basic functionalities like increasing/decreasing the amount of rune.
- Since it extends the Item class, it has all the functionalities from the class. The Rune object can be picked up by an actor using the PickUpAction and DropUpAction, once its portability is toggled to true. By default the portability is set to false (because player should not drop it), when player dies, its portability is toggled and will be dropped to the ground.
- The Rune object also implements the Resettable interface, which allows the object to be reset to its original state when the game is reset. When the game is reset, the Rune object is removed from the map if it has been reset too many times.This functionality can be used to control the states of rune after the player dropped it to the ground.

REQ 3

UML


engine
Item
PickUpAction
Action
Ground
Location
Actor
game
items
FlaskOfCrimsonTears
«Enum»
ItemUsage
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends
<<creates>>
<<Uses>>
actions
ConsumeAction
RecoverRuneAction
1
recovers
RestAction
DeathAction
<<Uses>>
<<Uses>>
reset
ResetManager
«interface»
Respawnable
«interface»
Resettable
environments
SiteOfLostGrace
Extends
TheFirstStep
<<Uses>>
<<Uses>>
<<Uses>>
<<Uses>>
<<Uses>>
<<Uses>>
runes
Rune
1
gameactors
players
<<abstract>>
Player
enemies
<<abstract>>
Enemy
<<Uses>>
<<creates>>
<<Creates>>

Sequence diagram - REQ3 - Rest Action



Design changes from Assignment 1:

1. No more **ResetAction** class, as it would have been redundant based on my implementation below
2. All earlier **dependencies from** the Resettables to the **ResetAction** class are now **towards** the **ResetManager**.
3. No **RespawnAction** class, as it is not an action made by choice; instead, it should be triggered for all things that can respawn automatically when they die. So we have an **Interface Respawnable** instead.

Flask Of Crimson Tears:

FlaskOfCrimsonTears is a game item that can be consumed to heal the player's hit points. The class extends the Item class and implements the Resettable interface. The item has various attributes, such as the maximum number of times it can be used, the amount it heals, and the number of times it has been used. It also has methods to update and reset the consumed status.

By separating the functionality of the FlaskOfCrimsonTears item into its class, the code follows the Single Responsibility Principle (SRP) of SOLID. This means that each class has a single responsibility or job and that any changes to the item's behaviour are isolated to this class only. This also makes the code easier to understand and maintain.

The code also follows the Open-Closed Principle (OCP) of SOLID by allowing the item to be easily extended or modified without changing the existing code. For example, new item properties can be added by adding new attributes to the class or new capabilities to the ItemUsage enum without changing the existing code.

Additionally, the code implements the Liskov Substitution Principle (LSP) by extending the Item class, a pre-existing class in the codebase, without changing its behaviour. This means that objects of the FlaskOfCrimsonTears class can be used instead of Item objects without issues.

Finally, the code follows the Interface Segregation Principle (ISP) of SOLID by implementing the Resettable interface, which defines a single reset method. This means that the FlaskOfCrimsonTears class only needs to implement the reset method and not any other methods from the Resettable interface, making it easier to maintain and less prone to errors.

In summary, implementing the FlaskOfCrimsonTears item meets SOLID principles by following the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, and Interface Segregation Principle.

Reset on Rest:

The reset feature has been implemented to make it extensible for future resettables. To achieve this, we have created a Resettable interface which defines the behaviour for resetting a component or system. Any class that implements this interface can be reset using the reset feature.

In addition to the Resettable interface, we have created a ResetManager class which maintains a list of all the resettable components and provides a single access point for resetting them. This design allows new resettables to be added to the system without modifying the reset feature code itself, thus making it more extensible.

To further enhance the extensibility of the reset feature, we have made it possible for resettables to register themselves with the ResetManager class. This means that new resettables can be added to the system dynamically without requiring any modifications to existing code.

Overall, the reset feature has been designed with extensibility, allowing it to be easily adapted and expanded as new resettables are added to the system. Using the Resettable interface and the ResetManager class, we have created a flexible and robust reset feature that can accommodate future changes and additions to the system.

The Resettable interface has following advantages in terms of SOLID principles:

- Single Responsibility Principle (SRP): The Resettable interface defines the behaviour for resetting a component or system. This helps to keep the interface simple and focused on a single task.

- Open-Closed Principle (OCP): The ResetManager class is designed to be open for extension but closed for modification. It achieves this by maintaining a list of Resettable objects and providing a single access point for resetting them. This design allows new resettable's to be added to the system without modifying the ResetManager code.

- Liskov Substitution Principle (LSP): Any class that implements the Resettable interface can be used interchangeably with other Resettable objects. This means that the behaviour of the ResetManager class is consistent regardless of the type of Resettable object being reset.

- Interface Segregation Principle (ISP): The Resettable interface is designed with only the necessary methods for resetting a component or system. This helps prevent clients from being forced to implement methods they do not need.

- Dependency Inversion Principle (DIP): The ResetManager class depends on the Resettable interface rather than concrete implementations of Resettable objects. This allows the ResetManager class to work with any object that implements the Resettable interface, promoting flexibility and extensibility.

Pros:

- □ Simplicity: The Resettable interface focuses on a single task, making it easier to understand and maintain.

- □ Consistency: The Liskov Substitution Principle ensures that any class that implements the Resettable interface can be used interchangeably, providing consistent behaviour for the ResetManager class.
- □ Separation of Concerns: The Interface Segregation Principle helps prevent clients from being forced to implement unnecessary methods, promoting better separation of concerns.
- □ Dependency Inversion: The Dependency Inversion Principle allows the ResetManager class to work with any object that implements the Resettable interface, promoting flexibility and modularity

Cons:

- □ Complexity: The design of the reset feature could be seen as more complex than a simpler approach, making it harder to understand for developers unfamiliar with it.
- □ Potential for misuse: While the Resettable interface is focused on a single task, there is a risk that it could be misused by developers who try to implement too much functionality in a single class.
- □ Maintenance: While the design of the reset feature promotes extensibility, it could require more maintenance over time as new resettables are added to the system.

.

Respawning:

The respawn method in the Player class is designed to reset the player's hit points to their initial value, which is the player's maximum hit points when it is first created. This is accomplished by calling the resetMaxHp method inherited from the Actor class.

The reset method is called from the respawn method whenever the player dies or rests at the Site of Lost Grace. This method is responsible for resetting the player's health and returning a string message to indicate that the player's health has been reset.

The Player class implements the Respawnable interface and defines the respawn method. This interface is used to mark actors that can be respawned in the game.

The Resettable interface is also implemented by the Player class, which is used to register the player with the ResetManager singleton class. The ResetManager class is responsible for registering all resettable objects in the game and resetting them to their initial state when the game is reset.

Pros:

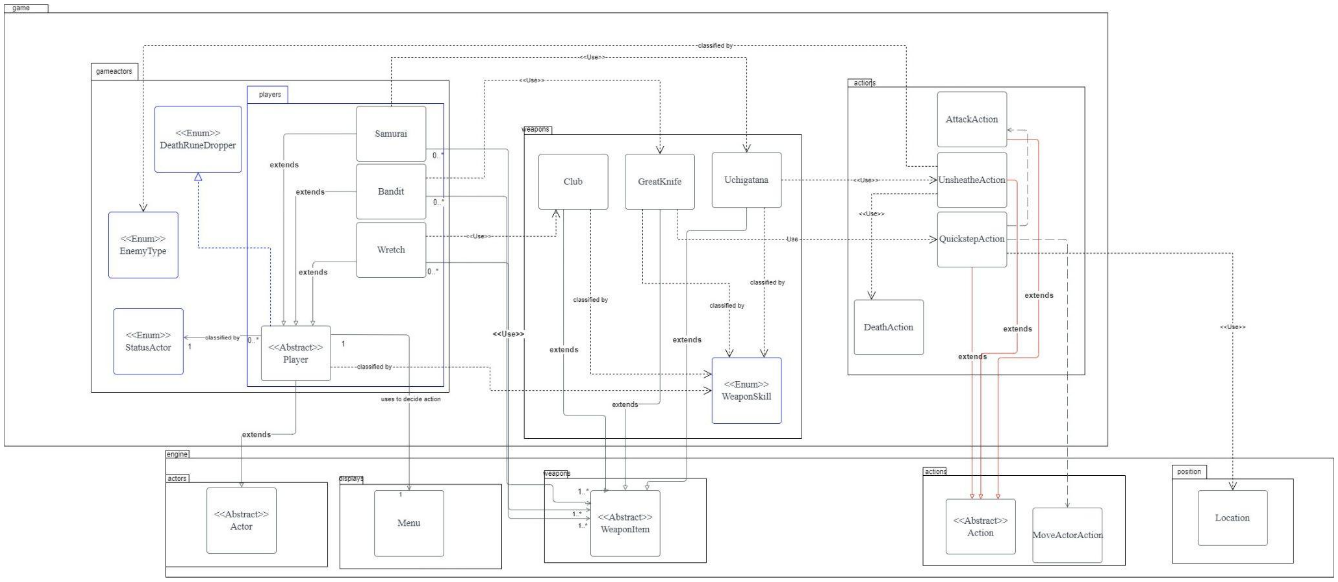
- □ The code follows the SOLID principles, making it easier to understand, maintain, and extend over time.
- □ The Respawn class is simple and easy to use, with a clear purpose and limited responsibility.

Cons:

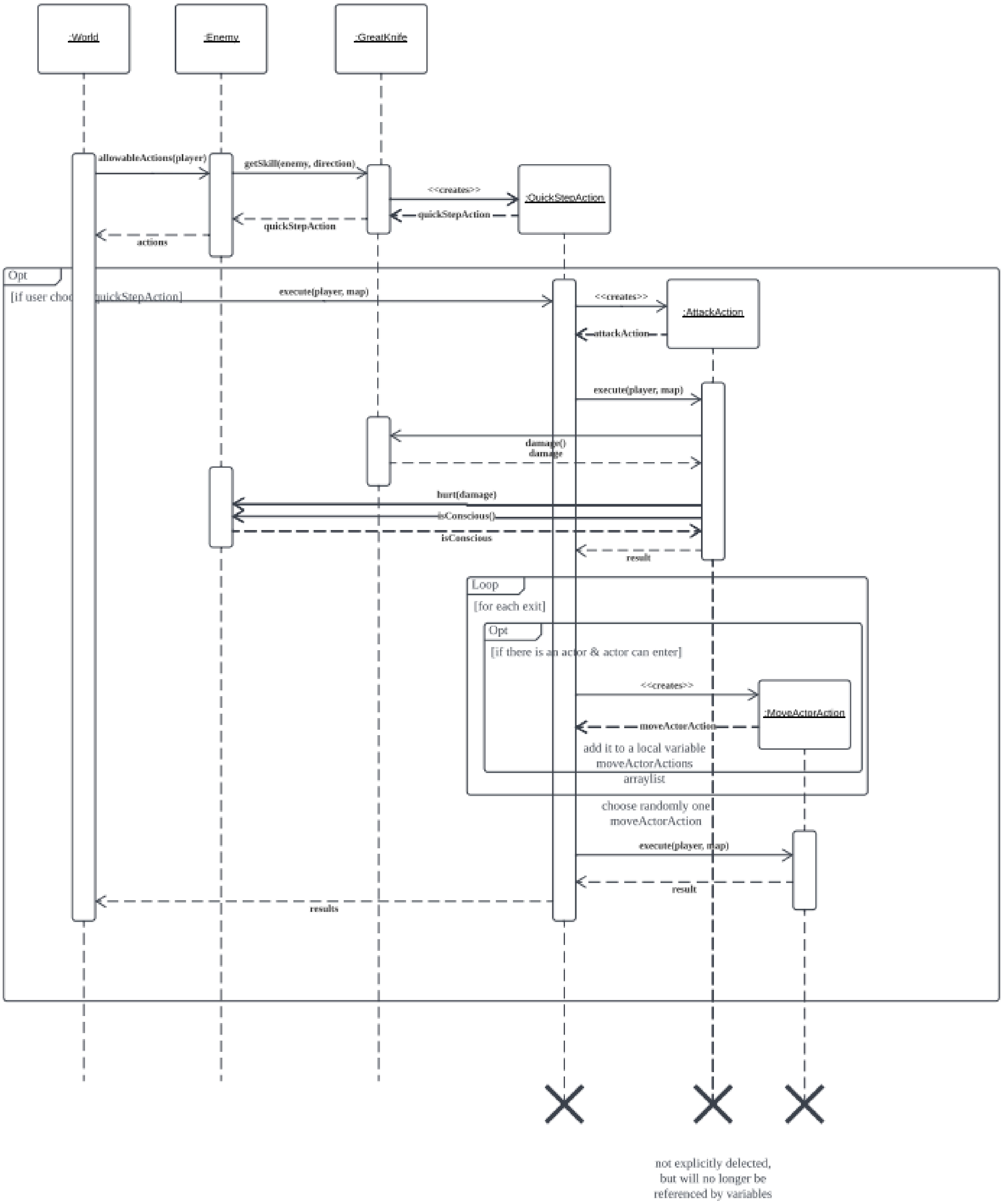
- □ The Respawn class does not handle error checking or exception handling, which could lead to unexpected behaviour if errors occur.

REQ 4:

UML



Sequence diagram (for QuickStepAction)


:World
:Enemy
:GreatKnife
allowableActions(player)
getSkill(enemy, direction)
<<creates>>
:QuickStepAction
quickStepAction
quickStepAction
actions
Opt
[if user choo quickStepAction]
execute(player, map)
<<creates>>
:AttackAction
attackAction
execute(player, map)
damage()
damage
hurt(damage)
isConscious()
isConscious
result
Loop
[for each exit]
Opt
[if there is an actor & actor can enter]
<<creates>>
:MoveActorAction
moveActorAction
add it to a local variable moveActorActions arraylist
choose randomly one moveActorAction
execute(player, map)
result
results
not explicitly delected, but will no longer be referenced by variables

Design Goals:

1. Implement a player
2. Implement combat archetypes of the player
3. Implement weapons and corresponding actions for each archetypes

Design changes from Assignment 1:

1. No significant design changes from the previous one. We ensured to add relationships/classes missing in Assignment1. (dependency relationship from each combat archetype to weapon, dependency relationship from QuickStepAction to Location, and dependency relationship from UnsheatheAction to DeathAction.)

Implementation Details - Player

- Player extends Actor and implements. The Player class has a menu, a ResetManager instance, a Rune instance, and a SiteOfLostGrace instance as attributes.
    - The class has a constructor that initializes the Player's hit points and adds various capabilities such as IS_PLAYER, CAN_RESPAWN, and CAN_REST.
    - The playTurn() method handles the player's turn in the game, displaying the player's status and menu, and allowing the player to take an action. In addition to possible targeted attacks/other actions that can be made to objects around, if the player has a weapon that can make area attack, this attack action will be added into actions in this method.
- Player implements Resettable interface. The reset() method resets the player's health when they die or rest.
- The class also has unique methods: 1) increase\decrease player's rune count, 2) check for the existence of enemies around the player's location, 3) get the player's previous location. The third method will be used to test if the player has moved or not inside the AttackBehaviour class.
- Player also implements the getDeathRune() method from the DeathRuneDroppper interface, which returns a Rune object that the player drops upon death.

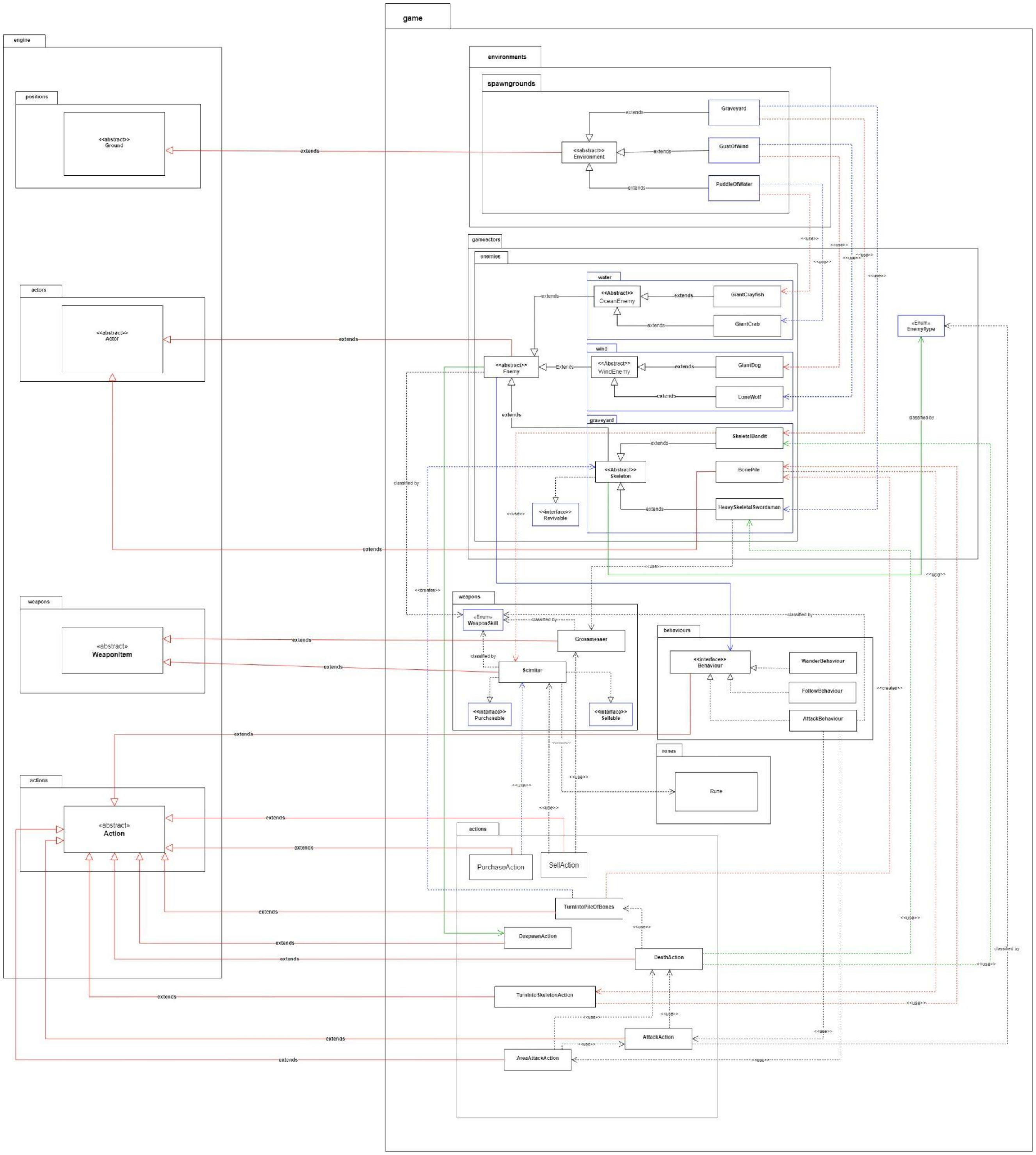
Implement Details - QuickStep

- The QuickStepAction class has two constructors, one that takes a weapon, target, and direction as parameters, and one that takes only target and direction (which is for intrinsic weapon).

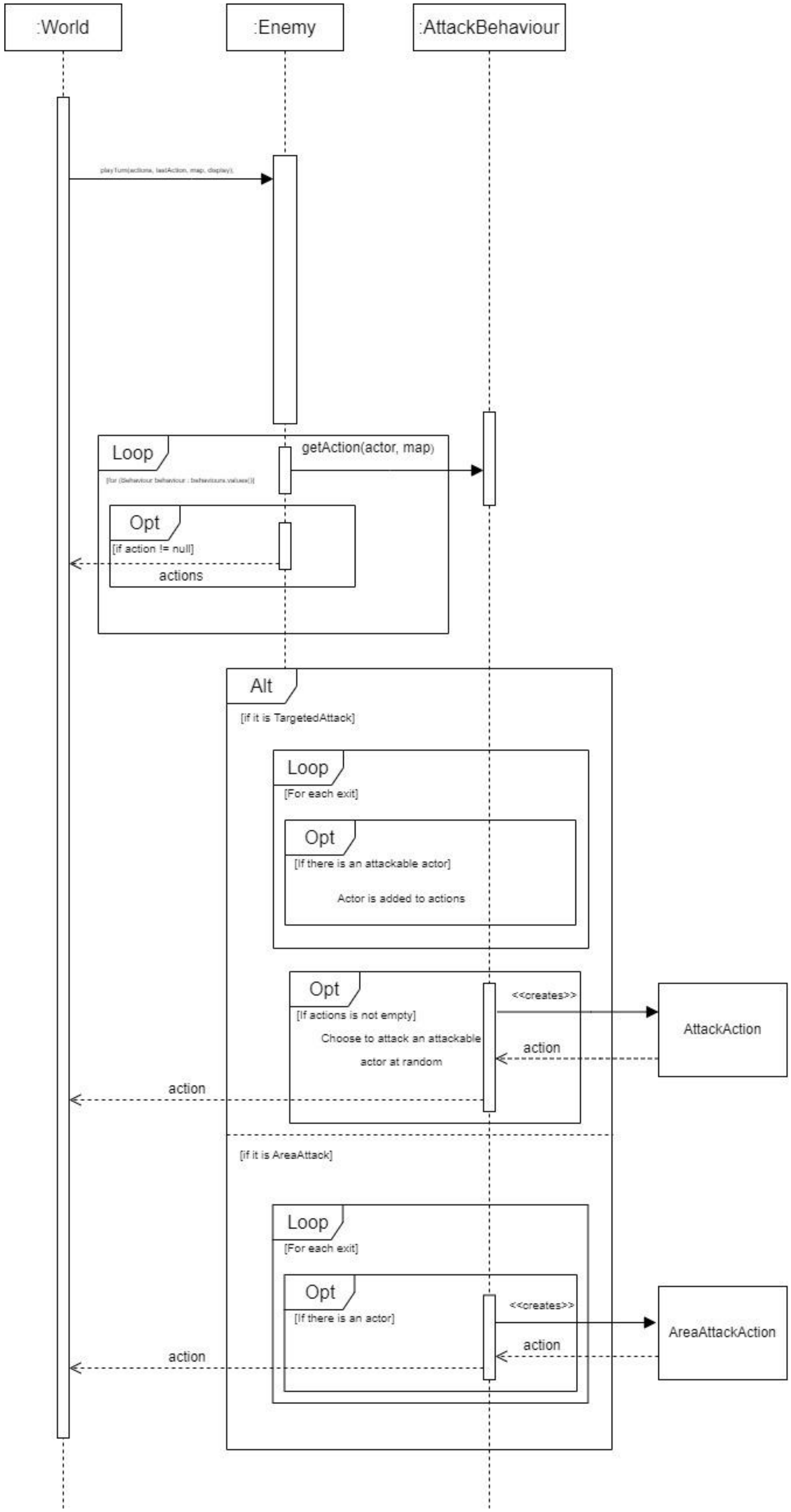
- The execute method make use of an AttackAction and a MoveActorAction internally. It first attacks a particular target specified through the constructor. Then, it traverses thorough the exits to find available positions the attacker can move to. If he can make a move, we instantiate a MoveActorAction. Then, if it have created at least one MoveActorAction, it will randomly choose one of them and is executed.

REQ5:

UML


engine
positions
<<abstract>>
Ground
actors
<<abstract>>
Actor
weapons
«abstract»
WeaponItem
actions
«abstract»
Action
game
environments
spawngrounds
<<abstract>>
Environment
Graveyard
GustOfWind
PuddleOfWater
gameactors
enemies
water
<<Abstract>>
OceanEnemy
GiantCrayfish
GiantCrab
wind
<<Abstract>>
WindEnemy
GiantDog
LoneWolf
graveyard
SkeletalBandit
<<Abstract>>
Skeleton
BonePile
HeavySkeletalSwordsman
<<interface>>
Revivable
<<abstract>>
Enemy
Extends
«Enum»
EnemyType
classified by
weapons
«Enum»
WeaponSkill
Grossmesser
Scimitar
<<interface>>
Purchasable
<<interface>>
Sellable
behaviours
<<interface>>
Behaviour
WanderBehaviour
FollowBehaviour
AttackBehaviour
runes
Rune
actions
PurchaseAction
SellAction
TurnIntoPileOfBones
DespawnAction
DeathAction
TurnIntoSkeletonAction
AttackAction
AreaAttackAction
extends
<<use>>
<<creates>>
classified by

Sequence diagram - REQ5 - AttackBehaviour



Environments:
Has the same implementation as REQ1, but checks for east and west side of the map.
Enemies mentioned in REQ1 spawn on the west side of the map

On the east side:
The Graveyard (SkeletalBandit), GustOfWind (GiantDog), and PuddleOfWater(GiantCrayfish) classes are subclasses of the Environment abstract class in a game. They represent specific types of game environments where enemies can spawn. Each class overrides the spawn method to randomly generate and return an instance of one or two types of enemy objects.

Gameactors:

Enemies:

Enemy (abstract class):
Has the same implementation as REQ1.

Graveyard:
Skeleton:
An abstract class that extends Environment.

SkeletalBandit:
The HeavySkeletonSwordsman class extends the Skeleton abstract class. It turns into a PileOfBones when it dies, can be be revived back. It overrides the revive method from the parent class to return a new instance of itself, and it adds a Grossmesser weapon to its inventory in the constructor. Revivable interface is implemented here.

PileOfBones:
Has the same implementation as REQ1.

Wind:
WindEnemy:
An abstract class that extends Environment.

GiantDog:
The LoneWolf class extends WindEnemy abstract class. The LoneWolf has an intrinsic weapon bite which allows it to deal damage to its enemies.

Water:
OceanEnemy:
An abstract class that extends Environment.

GiantCrayfish:
A class named GiantCrab is made that extends the OceanEnemy class that can perform a slam attack. The GiantCrab constructor calls the constructor of its parent class and sets the name, display character, hit points, and maximum damage of the crab.

The getIntrinsicWeapon method of GiantCrab returns an instance of IntrinsicWeapon class, which is a weapon that is built into the creature's anatomy and does not require the creature to wield a separate object.

Weapons:

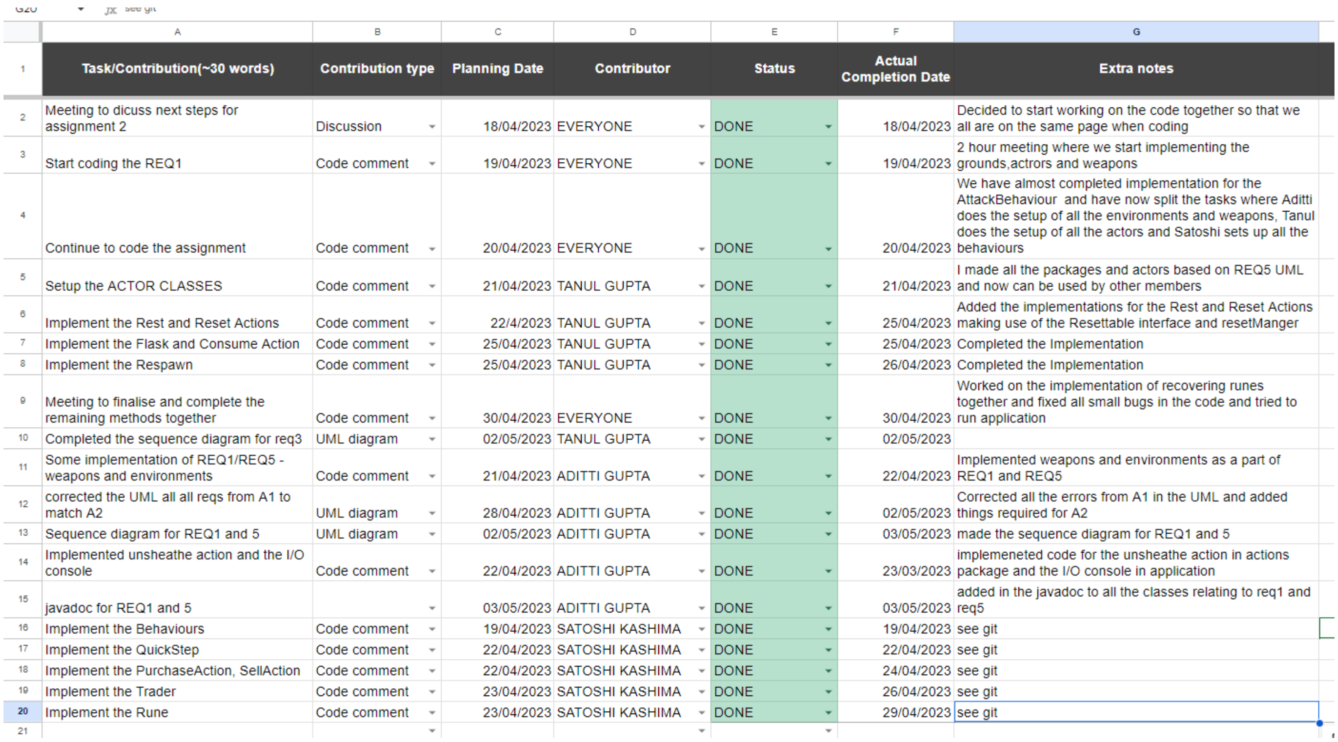
Scimitar:
The class Scimitar is a subclass of the WeaponItem class. It is a simple weapon that can be used to attack an enemy. The Scimitar implements the Purchasable and Sellable interface, which defines a method to get the selling price of the weapon. It also has a capability for AreaAttckAction when spinning attack is prompted.

Has the same pros and cons as Grossmesser.

Behaviours:
Has the same implementation as REQ1.

Actions:
Has the same implementation as REQ1.

Contribution log:



Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes
Meeting to dicuss next steps for assignment 2	Discussion	18/04/2023	EVERYONE	DONE	18/04/2023	Decided to start working on the code together so that we all are on the same page when coding
Start coding the REQ1	Code comment	19/04/2023	EVERYONE	DONE	19/04/2023	2 hour meeting where we start implementing the grounds,actrors and weapons
Continue to code the assignment	Code comment	20/04/2023	EVERYONE	DONE	20/04/2023	We have almost completed implementation for the AttackBehaviour and have now split the tasks where Aditti does the setup of all the environments and weapons, Tanul does the setup of all the actors and Satoshi sets up all the behaviours
Setup the ACTOR CLASSES	Code comment	21/04/2023	TANUL GUPTA	DONE	21/04/2023	I made all the packages and actors based on REQ5 UML and now can be used by other members
Implement the Rest and Reset Actions	Code comment	22/4/2023	TANUL GUPTA	DONE	25/04/2023	Added the implementations for the Rest and Reset Actions making use of the Resettable interface and resetManger
Implement the Flask and Consume Action	Code comment	25/04/2023	TANUL GUPTA	DONE	25/04/2023	Completed the Implementation
Implement the Respawn	Code comment	25/04/2023	TANUL GUPTA	DONE	26/04/2023	Completed the Implementation
Meeting to finalise and complete the remaining methods together	Code comment	30/04/2023	EVERYONE	DONE	30/04/2023	Worked on the implementation of recovering runes together and fixed all small bugs in the code and tried to run application
Completed the sequence diagram for req3	UML diagram	02/05/2023	TANUL GUPTA	DONE	02/05/2023	
Some implementation of REQ1/REQ5 - weapons and environments	Code comment	21/04/2023	ADITTI GUPTA	DONE	22/04/2023	Implemented weapons and environments as a part of REQ1 and REQ5
corrected the UML all all reqs from A1 to match A2	UML diagram	28/04/2023	ADITTI GUPTA	DONE	02/05/2023	Corrected all the errors from A1 in the UML and added things required for A2
Sequence diagram for REQ1 and 5	UML diagram	02/05/2023	ADITTI GUPTA	DONE	03/05/2023	made the sequence diagram for REQ1 and 5
Implemented unsheathe action and the I/O console	Code comment	22/04/2023	ADITTI GUPTA	DONE	23/03/2023	implemeneted code for the unsheathe action in actions package and the I/O console in application
javadoc for REQ1 and 5		03/05/2023	ADITTI GUPTA	DONE	03/05/2023	added in the javadoc to all the classes relating to req1 and req5
Implement the Behaviours	Code comment	19/04/2023	SATOSHI KASHIMA	DONE	19/04/2023	see git
Implement the QuickStep	Code comment	22/04/2023	SATOSHI KASHIMA	DONE	22/04/2023	see git
Implement the PurchaseAction, SellAction	Code comment	22/04/2023	SATOSHI KASHIMA	DONE	24/04/2023	see git
Implement the Trader	Code comment	23/04/2023	SATOSHI KASHIMA	DONE	26/04/2023	see git
Implement the Rune	Code comment	23/04/2023	SATOSHI KASHIMA	DONE	29/04/2023	see git