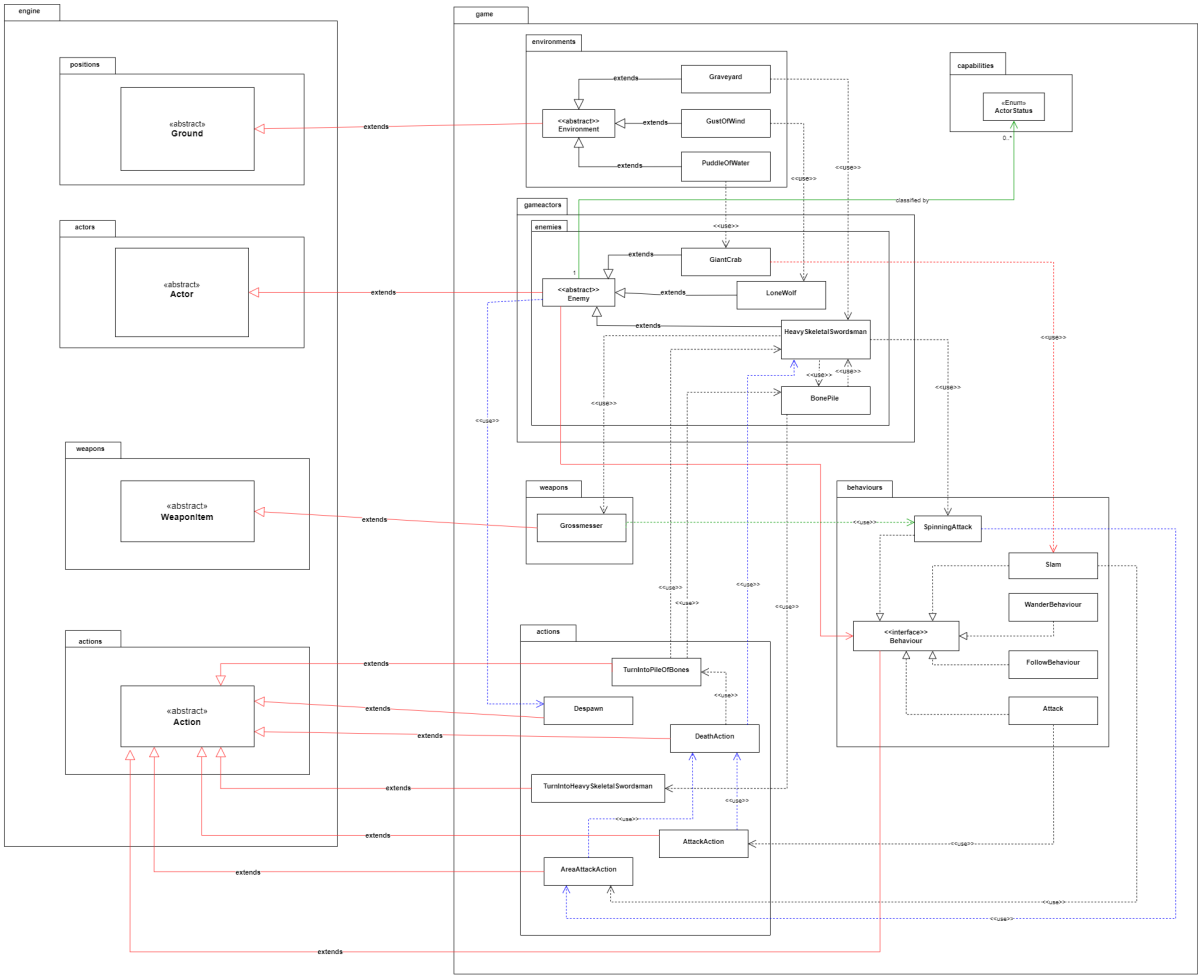
FIT2099
ASSESSMENT 1 - 15/04/2023
LAB7_GROUP8
Members: TANUL GUPTA, SATOSHI KASHIMA, ADITTI GUPTA

REQ1



Design Goal:

1. To implement all the different environments required
2. To implement all the different enemies required
3. To implement the weapon required

To achieve design goal 1:

- Using the abstract class Ground from engine, we extend an abstract class Environment, which extends three concrete classes Graveyard, Gust Of Wind and Puddle Of Water, all in the environments package. We are extending the 3 concrete classes from the Environment abstract class as they have similar properties.
- These environments are in dependencies with the enemies (seen in the next point).

To achieve design goal 2:

- Using the abstract class Actor from engine, we extend an abstract class Enemy, which further extends three concrete classes Heavy Skeletal Swordsman, Lone Wolf

and Giant Crab, all in the enemies package. We can do this because the enemies have the same basic characteristics.

- Heavy Skeletal Swordsman, Lone Wolf and Giant Crab spawn from Graveyard, Gust Of Wind and Puddle Of Water, respectively, showing dependency. These enemies can also Despawn (that is implemented in the action class).
- All behaviours implement the Behavior interface with the getAction method, which converts a behaviour into a corresponding Action object. These include WanderBehaviour (to wander around the map), FollowBehaviour (to follow the player when close) and Attack (to give damage using AttackAction that is implemented in the action class to a single enemy/player at a time).
    - Heavy Skeletal Swordsman could use the SpinningAttack behaviour.
    - Giant Crab could use the Slam behaviour
- Each of these enemies can attack other enemies and players, except Heavy Skeletal Swordsman (when using the SpinningAttack with the Grossmesser as seen below) and Giant Crab (when using Slam - behaviour class), which uses AreaAttackAction (damaging everyone in the area). To classify such enemy type, we have an enumeration ActorStatus.
- The Heavy Skeletal Swordsman has a special ability where when it is dead (using DeathAction), it turns into a Bone Pile that can be turned back into a Heavy Skeletal Swordsman (by using TurnIntoHeavySkeletalSwordsman and TurnIntoPileOfBones).
- When AttackAction is invoked (whether it's from the enemy or the player), the target will be hurt using the actor.hurt method. If the health becomes smaller than 0 due to this attack, DeathAction is instantiated. Therefore, there is a dependency from AttackAction to DeathAction.
- In comparison, if the target is HeavySkeletalSwordsman and its health is smaller than 0, we trigger the TurnIntoPileOfBones action, which replaces the HeavySkeletalSwordsman by PileOfBones

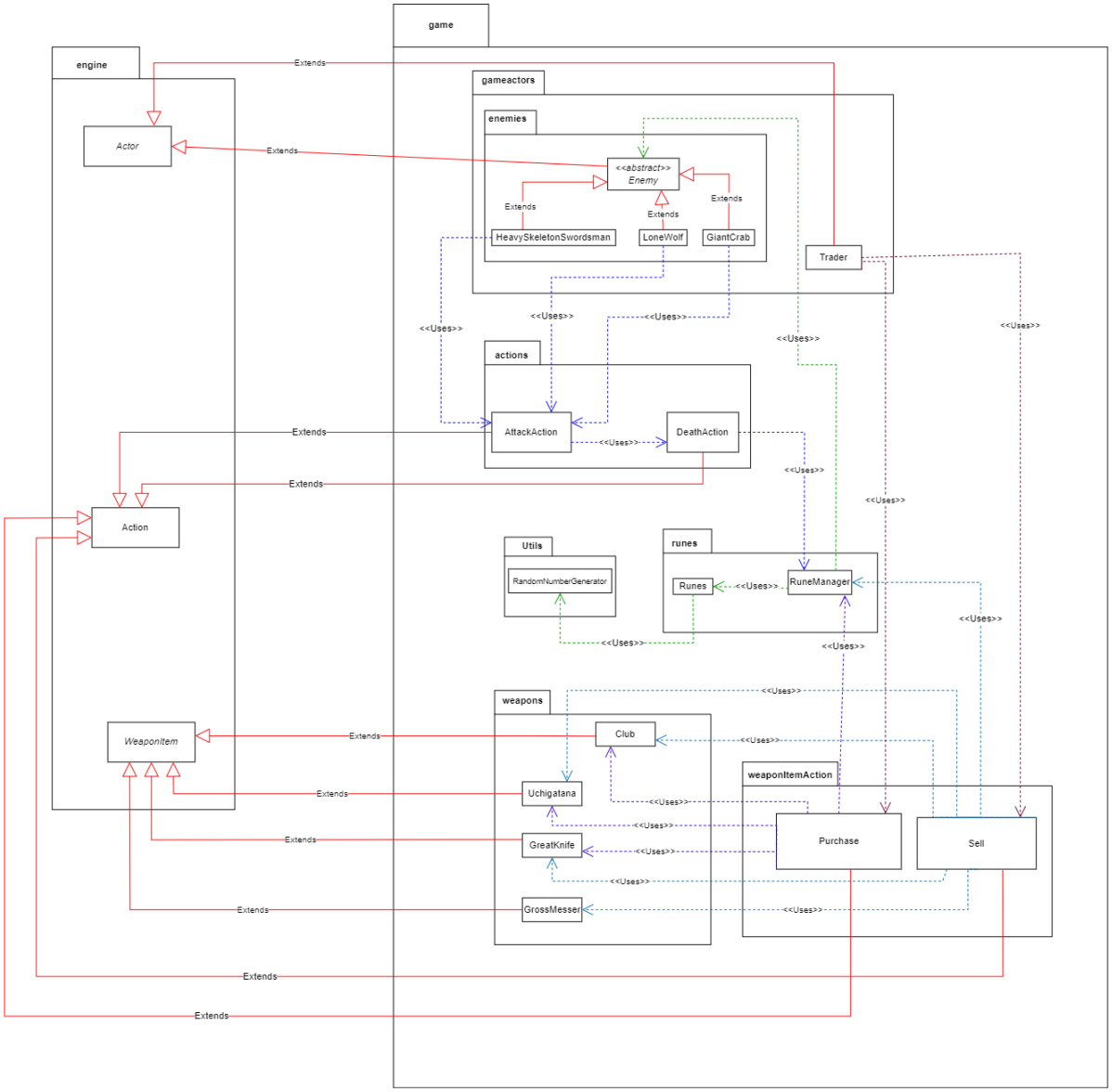
To achieve design goal 3:

- Using the abstract class WeaponItem from engine, we extend a concrete class called Grassmesser, a weapon used by the Heavy Skeletal Swordsman.
- This weapon uses AreaAttackAction to inflict damage on anyone in its surroundings, only when SpinningAttack is used (implemented in behaviours class)

Pros and Cons of this design:

- Consistent use of the Action class clarifies the game and reduces complexity even if we want to add more actions later on. However, this means that an action object must be instantiated every time there is an action, which does not happen if we just use methods to execute actions. Also, tweaking an existing action class might be difficult, since we cannot use interfaces to add functionality.
- By using abstract classes such as Enemy or Environment, we reduce a significant amount of code. However, this increases the number of classes and leads to multi-level inheritances (e.g. Enemy inherits Actor).
- Using Tick methods in different parts of the system enables access to the game map from locations, grounds, enemies, etc., at each turn, seamlessly. This enables the spawning of enemies in different environments, at each turn with some probability. However, calling tick methods of many methods at each iteration can be expensive. It also makes the debugging process harder.

REQ2



Design Goal:

1. The goal is to implement a system where 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). The Runes dropped will be random in a range specific to each of the enemies specified above.
2. The aim is to allow only the player to interact with the trader (Merchant Kale) to Purchase/Sell weapons to use the runes.

We have designed our system UML diagram to achieve this, using knowledge from Requirement 1 (**REQ1**).

To achieve design goal 1:

- First, we have used the Enemy abstract class we created in REQ1, which extends three concrete classes Heavy Skeletal Swordsman, Lone Wolf and Giant Crab.

- From the class's implementation, the player will conduct an attack using the AttackAction class that extends the Action class. The class allows us to conduct an attack and determine if the target actor of the attack is alive (conscious) or not.
- If an actor is not conscious (dead), we use the DeathAction class, extending the abstract action class. Here we conduct the actions that occur when an actor dies.
- If the attacker was the player, we use a method in RuneManager class, which will require the target (which would be any instance of the child of the Enemy abstract class that has died). We do this to get the range for the rune that will be dropped. We pass these range values to a method in the Rune class that will use the RandomNumberGenerator class to generate the random number of runes dropped in the range specific to the Enemy that died.

Up till here, we have achieved our design goal 1.

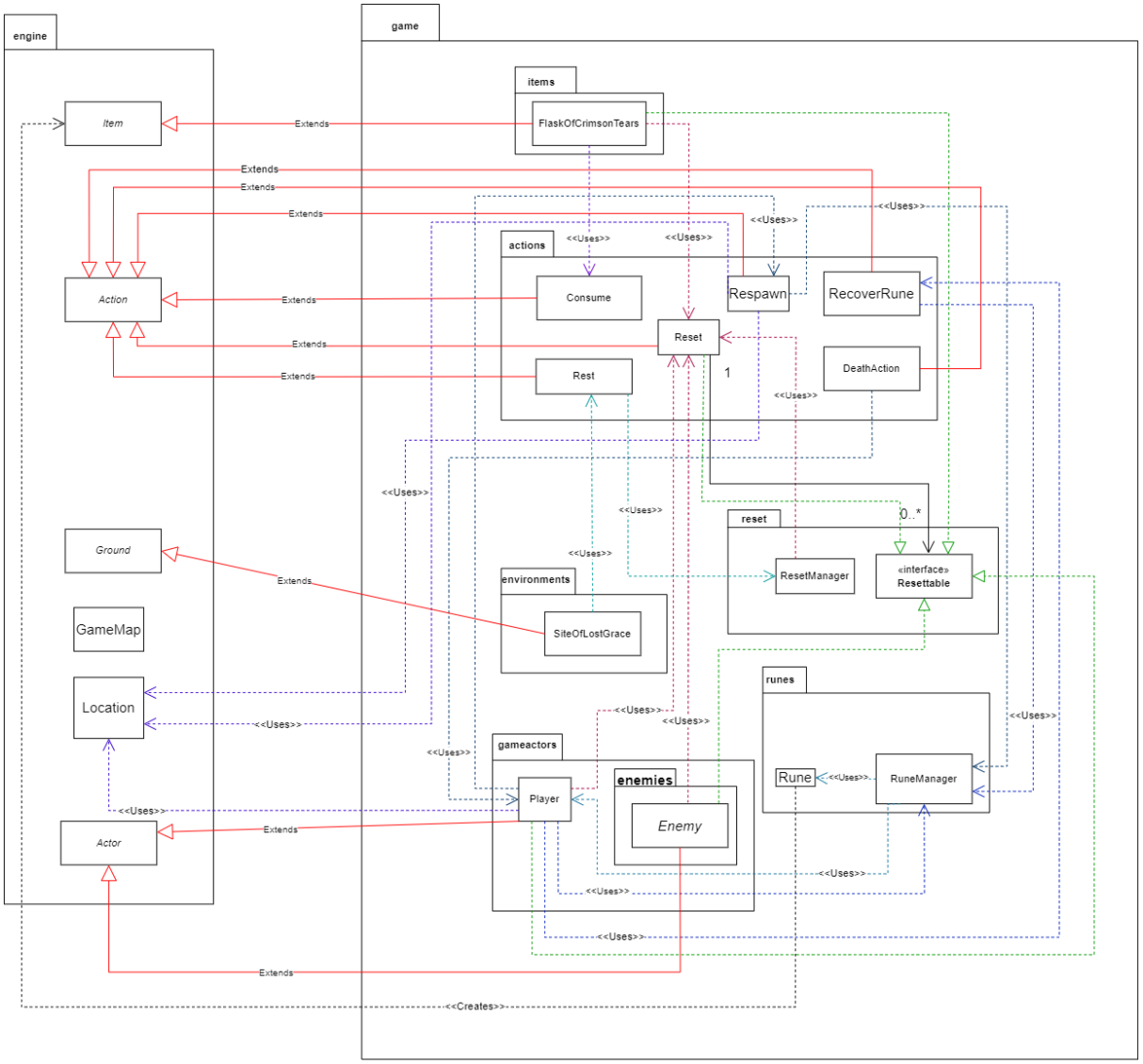
To achieve design goal 2:

- We have a new actor concrete class called Trader. It uses Purchase and Sell classes. Which one is called depends on what the player chooses when interacting with the trader. The Purchase class depends on all the weapons the player can buy from the trader. There are three dependencies, one each to Uchigatana, Great Knife and Club. There will be a call from purchase to the RuneManager to confirm the transaction; hence, there is a dependency.
- Similarly, for the Sell class, which contains dependencies to four weapons, these are the weapons the player can sell to the trader. Among the four weapons, three are the same. The only additional weapon is the concrete class Grossmesser. There will also be a call to the RuneManager class to manage the Players' runes by using a relevant method in the Rune class. If a valid transaction, allows the weapon to be added to the player's inventory.

Pros and Cons:

- The Rune class removes unwanted literals in the system when representing the game currency.
- The RuneManager class minimises the overall functionalities in Rune, which ensures that the class has a single responsibility.
- Separating the AttackAction and DeathAction enables the use of two actions independently in the future if needed.
- It may be better to have an intermediate class in the weapons package to use, but we will see during the implementation of the code if it is required.

REQ3



Design Goal:

1. Implement a healing Item called Flask Of Crimson Tears. Which has a limit to the number of times it can be consumed.
2. Implement a new ground called Site Of Lost Grace that allows the player to rest and respawn when killed.
3. Implement a way for the player to recover runes dropped when killed.

To achieve design goal 1:

- We create a concrete class called FlaskOfCrimsonTears that extends the Item abstract class, which in its allowableActions will have the method to heal the user by 250 points.
- There is a limit to the number of times it can be used:
    - Add a constant variable in FlaskOfCrimsonTears identifying the max consumption of the item, which is 2, and separately an instance variable holding the number of times consumed.

    - To act on consumption, we create a Consume class that extends the Action abstract class. Here we use the execute method to conduct the changes as required.

To achieve design goal 2:

- We extend the Ground abstract class from the engine and create a concrete class called SiteOfLostGrace which will have allowableActions of Rest.
- When a player uses the Rest action, it needs to call a static method in the ResetManager class called run() that calls the Reset class method reset that is to be implemented because it is part of the Resettable Interface.
- All classes that can be reset implement the Ressetable interface with a void method called reset(). Each class defines their own properties for reset and is called by the Reset class when the reset action needs to be performed. For this, the Reset class will have an ArrayList with Ressetable objects, loop through them, and call the reset methods.
- We have assumed all enemies up till now are resettable and have implemented the design that way

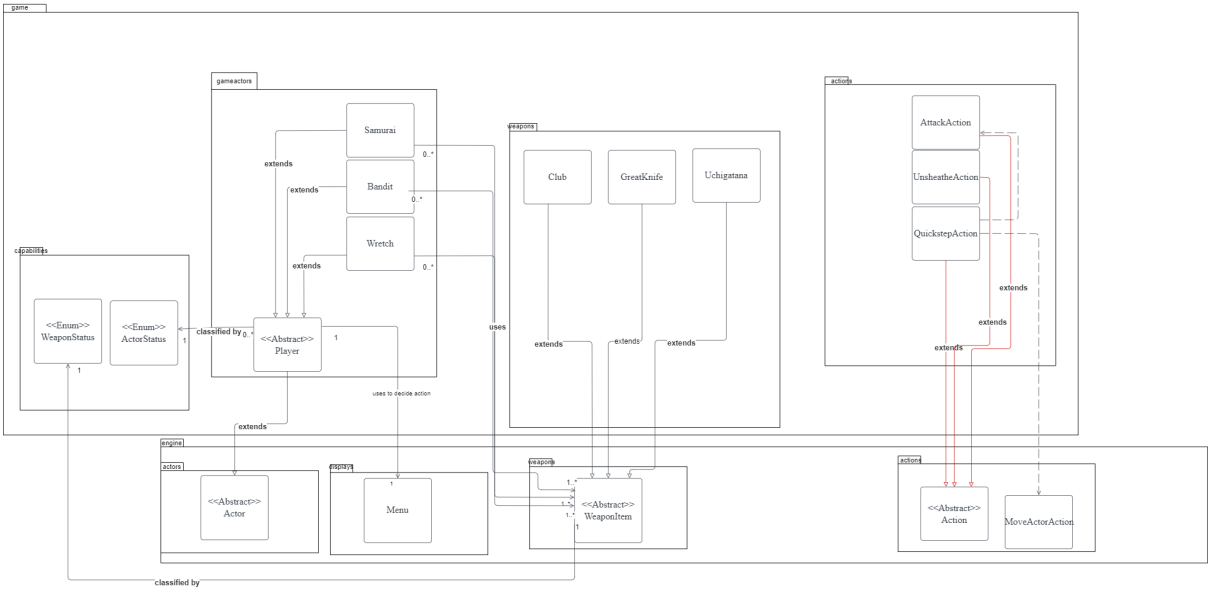
To achieve design goal 3:

- When a player dies, the DeathAction will see if the target actor is the player. The player class will use the Respawn action if it is the player. Thus we need the Respawn class that extends the Action abstract class. The execute() method will return the player to the last site of lost grace they rested at or where they spawned before they died (if not rested at any site of lost grace). It also drops runes at the spot they were at before they died: If the player "dies", their runes will be left on the location just before they died (e.g. if they are starting from (1,1) and then died at (1,2), the runes should appear in (1,1) instead of (1,2)). From the specifications for this, we call the RuneManager class to handle it with a method call to the Rune class, which will have a method to create an item of rune and display the Runes on the ground. The rune item will be terminated once picked up, or the player dies again.
- The player also has the ability to recover the runes they dropped. Hence, we need the RecoverRune class that extends the Action abstract class. This is an action when the player is at the location where there are Runes to pick up. Upon this action being called, the execute() method will call RuneManager to add the Runes to the player's current Runes.

Pros and cons:

Our design achieves the required design goals. While making use of the extensibility and the use of interfaces. We have reduced the overall number of dependencies. Even then, our design still does have a large number of dependencies between the classes.

REQ4



Design Goals:

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

To achieve design goal 1:

- We made the Player class abstract since the user must choose one of the archetypes available when playing the game
- The Player inherits Actor class. They share most of the functionalities, but Player requires Menu object to accept user input for action choice.
- The player is classified by an enumerable ActorStatus. It always has “hostile_to_enemy” status which can be used when using alloableActions of other actors.

To achieve design goal 2:

- We made a class for each archetype, Samurai, Bandit, and Wretch. They inherit the Player abstract class and have different constructors to instantiate their corresponding weapons, health, other attributes.
- Although each archetype player can only have one weapon initially, they can pick up and buy weapons from the trader. This is why the multiplicity is 0..” to 1..*.
- Since we need to apply different actions for the player depending on weapons available, each player type comes with WeaponStatus.

To achieve design goal 3:

- We made a class for each weapon: Club, GreatKnife, Uchigatana. This is because they all require different implementations.
- Each of them inherits WeaponItem class. There will not be any extra methods in these classes, but the attributes (such as damage, hitRate, verb, etc) will be

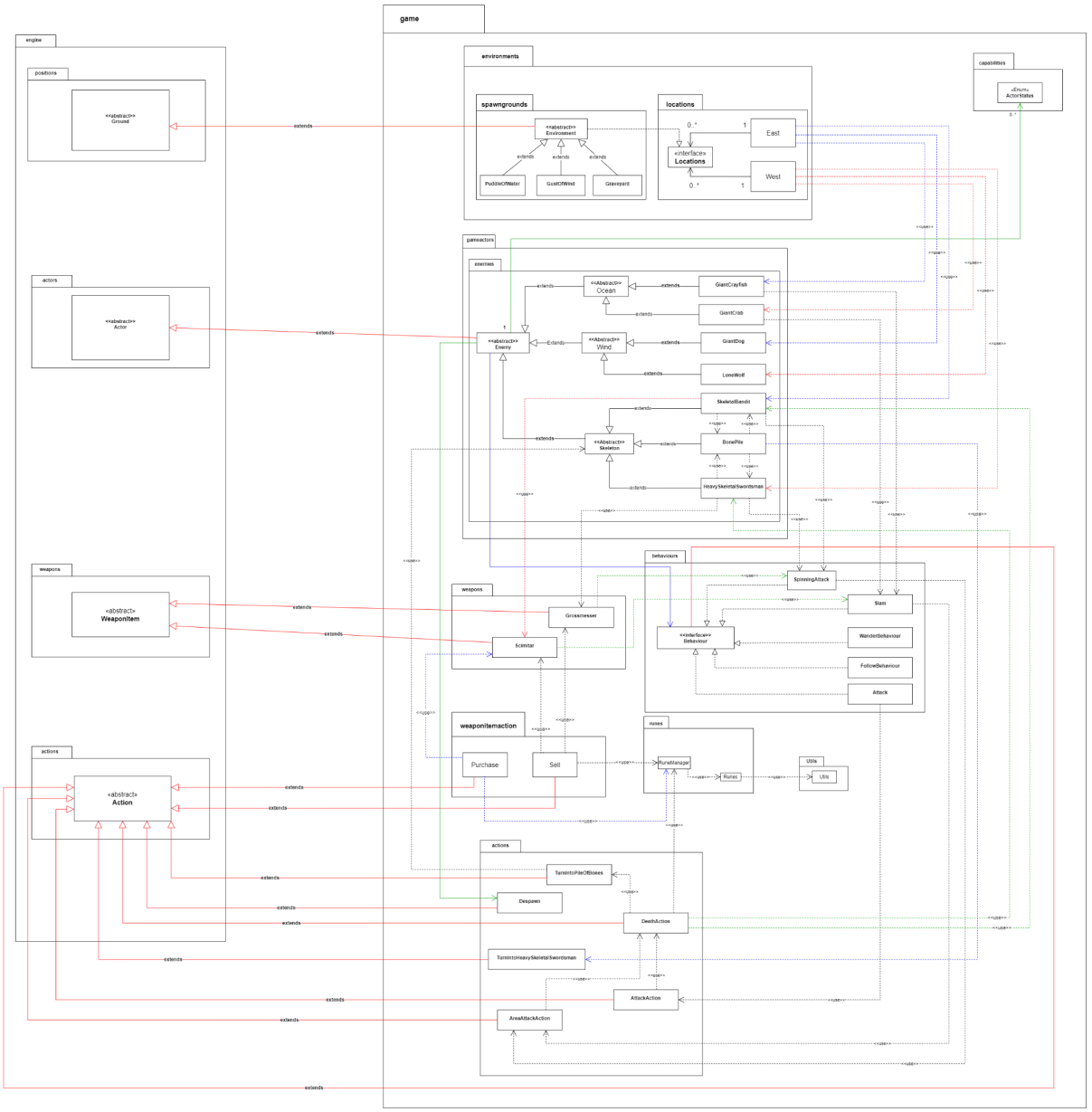
initialised with a constant value inside the constructor to avoid using literals many times.

- For each action that a player can make using these weapons, we made a class: AttackAction, UnsheathAction, QuickstepAction. They all inherit the Action abstract class. Since QuickstepAction can be considered as AttackAction + MoveActor, it has a dependency on both of them. This ensures that any actions related to weapons will be executed through the Action class, consistent with other parts of the system.
- Although not shown in the diagram, these actions will be instantiated in each enemy's allowableActions method. In the allowableActions(), we use the WeaponStatus to classify a given player type and instantiate action objects (described above) corresponding to that class. Hence,

Pros and Cons

- We reduce the number of literals present in the code by creating specific classes for each combat archetype.
- We combine the other two actions to form another action class, which reduces the amount of code required. This is an advantage of using Action classes. We can combine already existing action classes to create more complicated actions that we might need in the future.

REQ5



Design Goal:

1. Splitting the map into West and East
2. To implement more enemies required
3. To implement more weapons

To achieve design goal 1:

- The relationship for the Environment class is the same as REQ1 (with the same environments) but this class further shows dependency with the Locations interface, which contains West and East.
- Enemies spawn from the West side are Heavy Skeletal Swordsman, Lone Wolf and Giant Crab, and enemies from the East side are Skeletal Bandit Giant Dog and Giant Crayfish hence dependency is shown.

To achieve design goal 2:

- Using the abstract class Actor from engine, we extend an abstract class Enemy, which further extends to three abstract classes named Ocean, Wind and Skeleton.
    - Ocean extends two concrete classes Giant Crayfish and Giant Crab.
    - Wind extends two concrete classes Giant Dog and Lone Wolf.
    - Skeleton extends two concrete classes Heavy Skeletal Swordsman and Skeletal Bandit.
    - We do this because of their shared characteristics
- Heavy Skeletal Swordsman, Lone Wolf and Giant Crab spawn from Graveyard, Gust Of Wind and Puddle Of Water, respectively, on the West side of the map whereas Skeletal Bandit, Giant Dog and Giant Crayfish spawn from Graveyard, Gust Of Wind and Puddle Of Water respectively on the East side of the map and hence shows dependency. These enemies can also Despawn (implemented in the action class).
- They each also have certain behaviours, same as REQ1, the only difference being that the Heavy Skeletal Swordsman and Skeletal Bandit could use a spinning attack as a behaviour, and Giant Crab and Giant Crayfish could use a slam as a behaviour.
- Each of these enemies can attack other enemies and players, except Skeletal Bandit (when using the SpinningAttack with the Scimitar as seen below) and Giant Crayfish (when using Slam), which uses AreaAttackAction (damaging everyone in the area).
- The Skeletal bandit is similar to the Heavy Skeletal Swordsman, which has a special ability where when it is dead (using DeathAction), it turns into a Bone Pile that can be turned back into a Heavy Skeletal Swordsman (by using the action classes TurnIntoHeavySkeletalSwordsman and TurnIntoPileOfBones).
- DeathAction is also used when a player/enemy dies/is killed.

To achieve goal 3:

- A new weapon name Scimitar is added that extends from the abstract class WeaponItem from engine. This is used by the Skeletal Bandit.
- This weapon uses AreaAttackAction to inflict damage on anyone in its surroundings only when SpinningAttack is used (implemented in behaviours class)
- The Scimitar can be purchased and sold (using Sell and Purchase from the abstract class Action) using RuneManager (that uses Runes), whereas the Grossmesser can only be sold not purchased.

Pros and cons:

- We made new abstract classes to group the enemies, which can be efficient in reducing the number of codes, especially if we have more enemies in the future. However, this leads to multi-level inheritance and the performance of the game can decrease.

CONTRIBUTION LOGS:
LINK:
https://docs.google.com/spreadsheets/d/1XwBDR87s39Gtv3NWy94DF-U9vQUq9DwdsreGggonu3s/edit#gid=0

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes
First meeeting discussion	Discussion	26/03/2023	EVERYONE	DONE	26/03/2023	We had a 1 hour meeting to read the assignment specifications together. The team agreed to have another meeting on the 30th of March to digest the requirements.
Splitting Of Tasks	Discussion	30/03/2023	EVERYONE	DONE	30/03/2023	We had a meeting and we discussed Requirement 1 and then split the tasks amongst each other where TANUL: REQ 2, REQ 3, REQ 5, SATOSHI: REQ4, REQ5, ADITTI: REQ 1, REQ 5
List out all requirements for req 2	Brainstorm	03/03/2022	TANUL GUPTA	DONE	03/03/2023	Also had help and advice from Satoshi
List out all requirements for req 3	Brainstorm	03/03/2023	TANUL GUPTA	DONE	03/03/2023	
Design the UML Diagram for REQ 2	UML diagram	03/03/2023	TANUL GUPTA	DONE	12/04/2023	After making many drafts and commits to a branch in git, receiving help from consultations I completed the UML Diagram
Design the UML Diagram for REQ 3	UML diagram	08/03/2023	TANUL GUPTA	DONE	14/04/2023	After making many drafts and commits to a branch in git, receiving help from consultations I completed the UML Diagram
Brainstorm all the requirements for REQ 5	Brainstorm	03/04/2023	EVERYONE	DONE	03/04/2023	We brainstormed all the ideas we had for requirement 5
Design the UML for Req 5	UML diagram	03/04/2023	EVERYONE	DONE	14/04/2023	All of us worked together designing the UML, while having worked on our parts of the requirements and as we completed our requirements we kept updating REQ 5
Design Rationale		12/04/2023	TANUL GUPTA	DONE	14/04/2023	Completed the full design rationale for Requirements 2 and Req 3
Final discussion on important parts of the UML d	Discussion	15/04/2023	EVERYONE	DONE	15/04/2023	A small meeting on combat archtypes, pile of bones and some other implementation details
UML Diagram for req4	UML diagram	09/04/2023	SATOSHI KASHIMA	DONE	15/04/2023	Completed UML diagram for req4
UML Diagram for req2 (draft)	UML diagram	10/04/2023	SATOSHI KASHIMA	DONE	12/04/2023	A draft on UML diagram for req2
UML Diagram for req1 and 5 (draft)	UML diagram	11/04/2023	SATOSHI KASHIMA	DONE	13/04/2023	A draft on UML diagram for req1 and req5
List out all requirements for req 1	Brainstorm	03/07/2023	ADITTI GUPTA	DONE	03/03/2023	also discussed with other teammates
Design the UML Diagram for REQ 1	UML diagram	15/04/2023	ADITTI GUPTA	DONE	15/04/2023	Started the draft of REQ1 UML on 10/04/2023 but finished by 15/04/2023
Design Rationale		15/04/2023	ADITTI GUPTA	DONE	15/04/2023	Completed the full design rationale for REQ1 and REQ5

Link to the diagram:
https://drive.google.com/file/d/1ecWg7gSVsgvuXbBULQpVMe_mexnHGZmM/view?usp=sharing