AppliedSession21_Group8

FIT2099 Assignment 3: Further Implementation

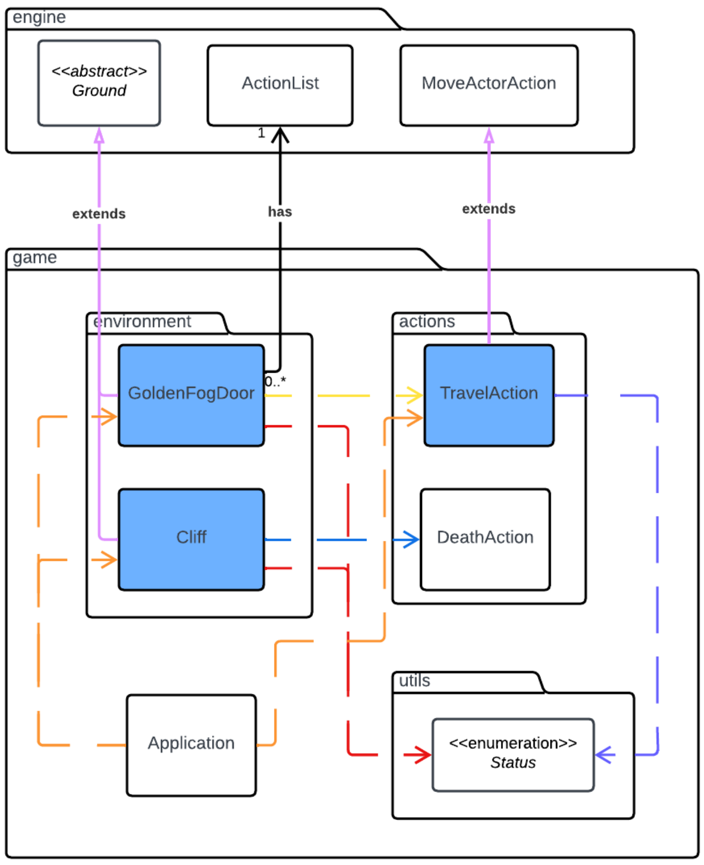
(modified by 21/5/2023)

Group member:

1.	Ying Qi Mah	32796765
2.	Kai Le Aw	32202490
3.	Jun Quan Ng	32722664

Assignment 3: UML diagram and Rationale

<u>Requirement 1</u>



Rationale:

Cliff

- This class extends the Ground abstract class in the engine package because the environment is ground.
- As Cliff uses the same attribute and method as ground, we inherit the ground class to avoid repetition (DRY).
- Cliff has dependency to DeathAction because player will be dead when step onto cliff
- Cliff has dependency to Status because it needs to check if a player can fall on a cliff.
- The benefit of this implementation is that we do not repeat the implementation that we have done for ground, and we believe that there is no downside for this implementation.

GoldenFogDoor

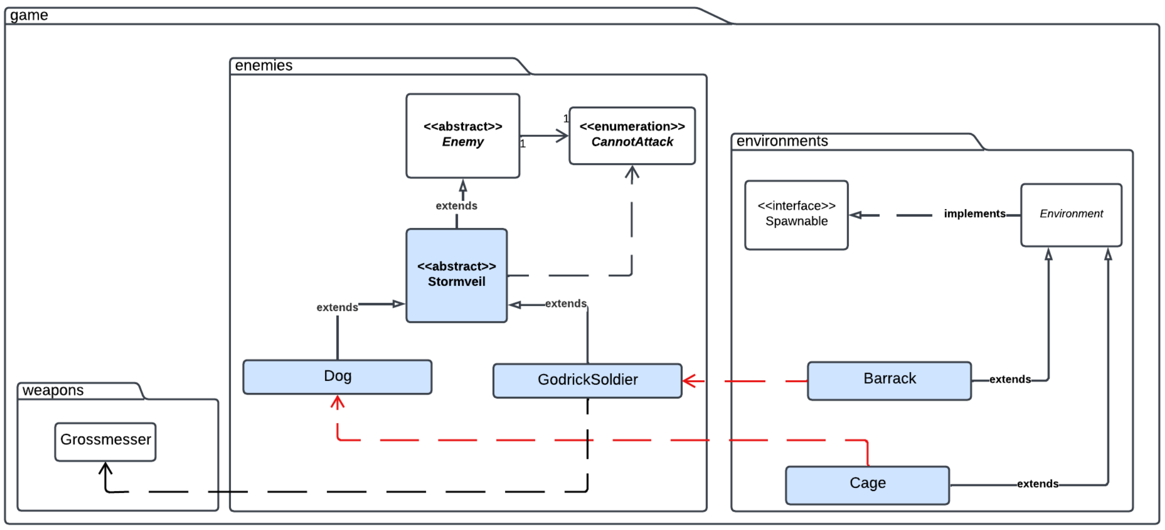
- This class extends the Ground abstract class in the engine package because the environment is ground.
- Since GoldenFogDoor uses the same attribute and method as ground, we inherit the ground class to avoid repetition (DRY).

- GoldenFogDoor has dependency to TravelAction because player can travel through different maps using GoldenFogDoor.
- GoldenFogDoor has association to ActionList because GoldenFogDoor has TravelAction.
- GoldebFogDoor has dependency to Status because it needs to check that a player can travel using GoldenFogDoor.
- The benefit of this implementation is we can easily extend it by adding different TravelAction to the door to allow user to travel to multiple locations. The downside is that we are travelling to a location, not to a door, hence we need to specify the exact location (x,y) that we are travelling to.

TravelAction

- It extends the MoveActorAction class as it is an action which can move actor around, this could prevent the repetition of code (DRY).
- It has dependency on Status because only player with travel status can perform this action.

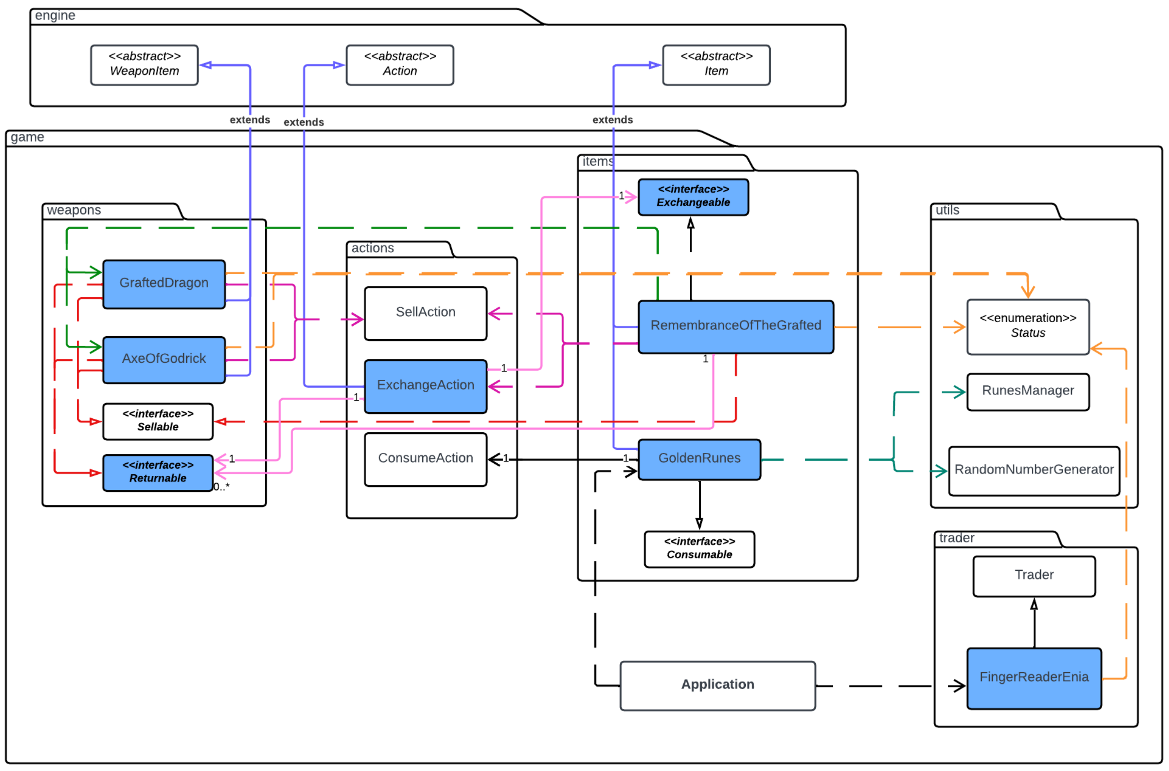
Requirement 2



Rationale:

- Stormveil abstract class is created to allow extension from Dog and Godrick Soldier classes. As both enemies are of the same type (CannotAttack each other). We decided to implement it this way.
- However, there could be an alternative way to implement Dog and GodrickSoldier classes. The alternative way is to let both subclasses extend directly to the abstract Enemy class that has been implemented before. This is because we could simply set their type (CannotAttack) in every subclass to reduce multiple abstract class layers.
- After long consideration, we still decided to use Stormveil abstract class in between subclasses and Enemy class. This is because it would be simpler for us to add more on enemies in Stormveil type as a game could have more and more enemies. Thus, this helps us to achieve DRY principle as we would not need to define it's CannotAttack type in every subclass again.
- Regarding Barrack and Cage, we designed and implemented the way as we have done in previous assignments. For every environment that can spawn enemies would extend to Environment abstract class.
- This is because it helps us to achieve DRY principle and we have ensured that every enemies' environment would definitely need to implement Spawnable's interface codes. Thus, we can finally ensure Barrack and Cage can spawn it's individual enemy within it's area.

Requirement 3



Rationale:
GoldenRunes

- It inherits the item as it is an item to follow, don't repeat yourself principle(DRY).
- It has an association on ConsumeAction because a player can consume it.
- It implemented the interface consumable because it can be consumed. The benefit is to allow other classes to use the method that a consumable has.
- It has dependency on RunesManager and RandomNumberGenerator because it need to add player runes with random amounts of 200-10000 runes.
- The benefit is we do not have to repeat the attributes and methods that an item already has.

RemembranceOfTheGrafted

- It inherits the item as it is an item to follow, don't repeat yourself principle(DRY). It implemented interface Exchangeable because it can be exchanged to traders.
- It implemented an interface Sellable because it can be sold to traders.
- It has dependency on sell action and exchange action because it can be sell and exchange
- It has association with returnable because it has a list of items which can be exchanged.
- It has dependency with status because it needs to check if the NPC can perform sell action or exchangeable action.
- It has dependency with GraftedDragon and AxeOfGodrick because they are the item in return of exchanging RemembranceOfTheGrafted
- The benefit is we do not have to repeat the attributes and methods that an item already has. The downside is that we are storing the GraftedDragon and AxeOfGodrick in the RemembranceOfTheGrafted instead of in the FingerReaderEnia, this could be good depends on how the game proceeds to expand. If only RemembranceOfTheGrafted can be exchanged for GraftedDragon and AxeOfGodrick, then it is can be easily extended.

Exchangeable

- It is created for items that can be exchange to implement and prevent multi-level inheritance. It ensures that classes that implement it can be exchange.
- The benefit is that it makes the game easier to extend as all items that can be exchange only need to implement this interface.

ExchangeAction

- It extends the Action class as it is an action, this could prevent the repetition of code (DRY).
- It is created because exchangeaction is a single action, so it follows the single responsibility principle.
- It has association with exchangeable and returnable because it needs to use these two items to complete exchange action.

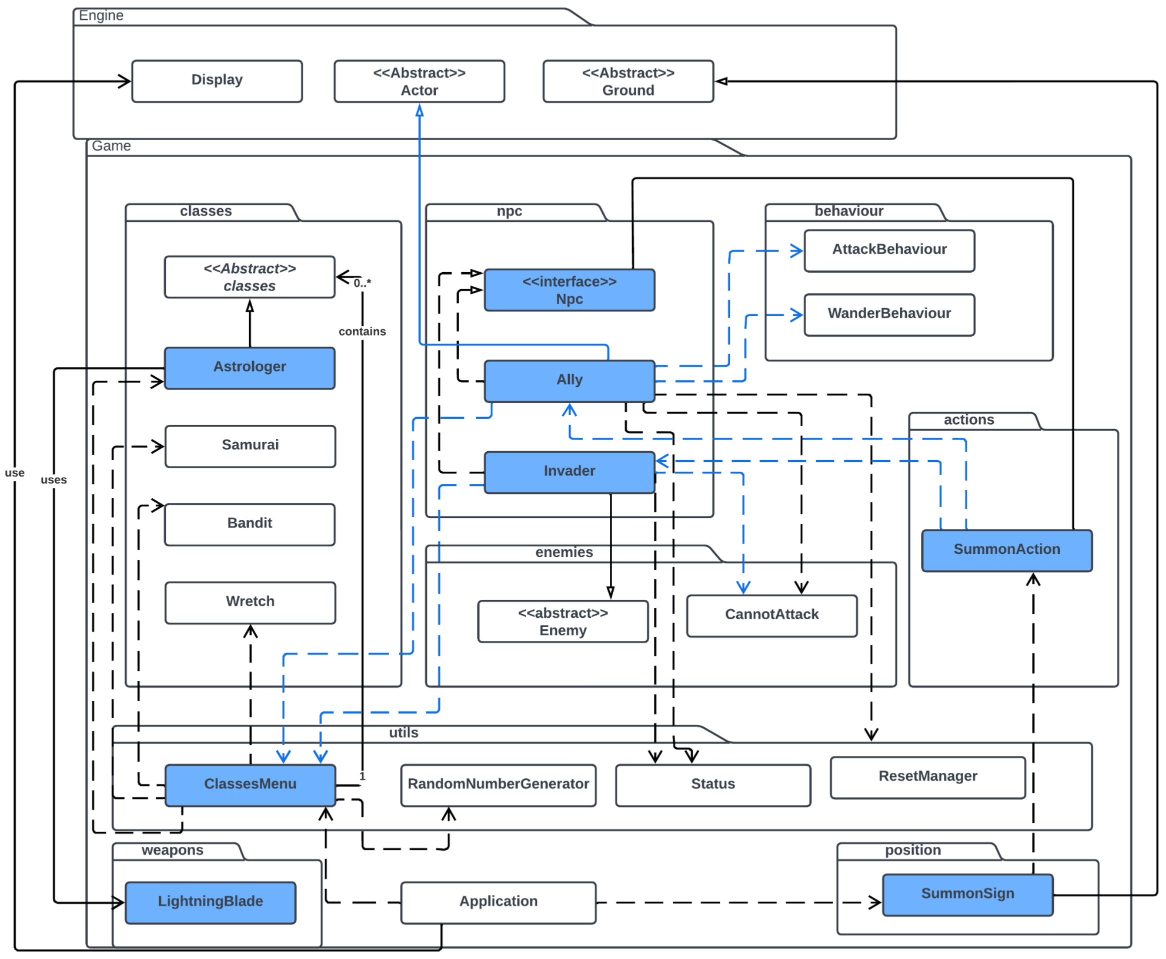
Returnable

- It is created for items that can be returned to implement and prevent multi-level inheritance. It ensures that classes that implement it can be returned after exchange.
- The benefit is that it makes the game easier to extend as all items that can be return only need to implement this interface.

GraftedDragon & AxeOfGodrick

- GraftedDragon & AxeOfGodrick are all WeaponItem, so they extend WeaponItem abstract class to prevent repetition(DRY).
- Both have dependency on sell action because it can be sell.
- Both implemented the interface Returnable because it can be returned by traders after exchange with an item.
- Both implemented an interface Sellable because it can be sold to traders.
- Both have dependency with status because they needed to check if the NPC can perform sell action.

Requirement 4



Design Rationale
Astrologer
- A new Astrologer Class is created that inherits classes abstract class, this is to prevent code repitition which follows DRY principle. The benefit is we do not need to repeat the implementation made for other classes.
- It has a LightningBlade as starting weapon (Requirement 5) hence there's an association.

ClassesMenu
- A new ClassesMenu class is created, it's function is to allow player to choose one of the classes to play with, and allow ally and invader to randomly choose a class.
- It has an association to classes as it provides a list of classes to be chosen from.
- It's has the responsibility to provide classes to player, ally and invader, hence it follows single responsibility principle.
- The benefit of this is that we don't need to repeat the code for each part where it needs classes choosing. It can also be easily extended by adding another class in the classes menu, hence it follows the open closed principle.

SummonSign
- A new SummonSign class is created that inherits Ground abstract class, this is to prevent code repitition which follows DRY principle. The benefit is we do not need to repeat the implementation made for other ground classes.
- It provides player the summon action for summoning Npcs.

SummonAction

- An action to randomly summon an ally or invader, it inherits the action abstract class as it is an action so it follows DRY principle. It is only responsible for summoning, hence it follows the Single Responsibility Principle too.
- It has a dependency on RandomNumberGenerator as it needs to randomly summon one of the ally and invader. It can be easily extended afterwards as it uses classes menu to choose a class that the ally and invader to spawn as.

Npc

- A new interface for non player character like ally and invader that can be summoned. It ensures that they have the method for Npc. The benefit of this is to prevent multi inheritance. The downside is that each classes has to repeat the same method. It can be easily extended as if we want to add more Npcs, we just need to implements this interface.

Invader

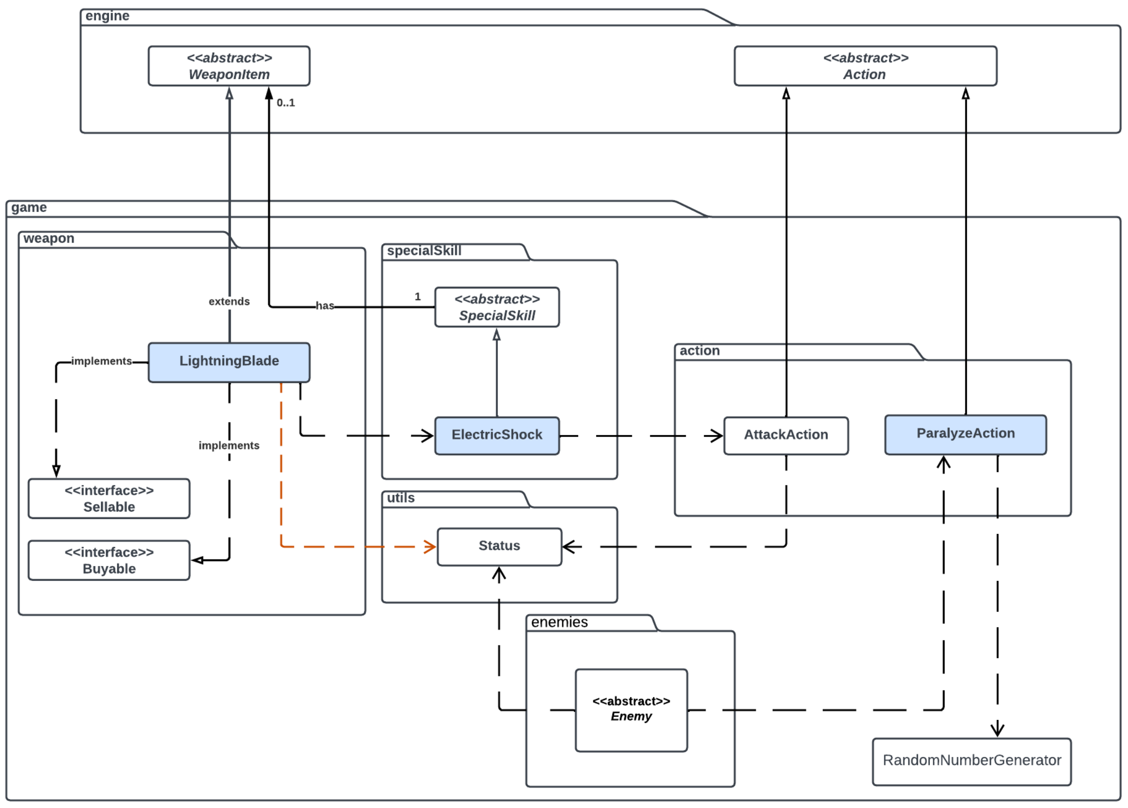
- A new class that inherits Enemy and implements Npc. It inherits Enemy because it has the same behaviour as enemies this is to prevent the repetition of code, which follows DRY principle.
- As Invader is an enemy but resetting algorithm is different, it overrides the reset function and implement its own one.

Ally

- A new class that inherits actor and implements Npc. It inherits actor because it is an actor, this is to prevent the repetition of code, which follows DRY principle. The benefit of this implementation is that it does not need to repeat the code for actors, but the downside is that it needs to rewrite what behaviour that it has, unlike invaders.

The alternative of this implementation is to create a new abstract class named friendly Npc, to differentiate the Npc that can attack player and those who cannot

Requirement 5



Rationale:

- To briefly summarise the creative requirement 5, a weapon LightningBlade is created that has a specialSkill named as ElectricShock that could deal damage to a single target enemy and apply a paralyze effect on the enemies for N rounds.
- ParalyzeAction will determine the number of rounds based on the RandomNumberGenerator.
- ElectricShock is a type of specialSkill that relates to AttackAction while ParalyzeAction is a type of new action to be carried out to put an enemy doing nothing for N rounds.
- As it is a new weapon, we would extend the WeaponItem class to achieve DRY principle as it shares common attributes and methods.
- ElectricShock will extend SpecialSkill (created in Assignment 2) to achieve DRY principle as it shares common attributes and methods. For example, ElectricShock is meant to be a single target special skill that will paralyse enemies. Therefore, it will need to extend AttackAction so as SpecialSkill has done (common methods).
- ParalyzeAction class is created as we desire to achieve the Single Responsibility Principle, it will only deal with methods that are related to the paralyze effect on Enemy. For every round, it will first check if an enemy has a paralyzing status effect. If it did have this effect, it will perform paralyzeAction as it uses RandomNumberGenerator to determine its probability to remove paralyze status (so it can finally move to the next round).
- However, the alternative way is to directly determine enemies' status (has a paralysing effect on it) within the Enemy class. Therefore, we would not necessarily create it as an Action class to just handle the probability of being paralyzed again in a different class.

- In the end, we still have decided that making it a single class to handle paralyze action is the best way of design and implementation due to the SOLID principles (Single Responsibility and Open Closed).
- There is another alternative way as well. Instead of creating a paralyzeAction, we could also implement it as paralyzeBehaviour as it works fine (it is an behaviour of cannot move). However, both ways actually performed the exact same thing and both are meant to reduce the code to its minimum as well. Thus, we picked ParalyzeAction as we preferred it to be an action (action taken by player onto enemy, not a natural behaviour).
- ParalyzeAction also applies the Open Closed principle as it will not affect other classes. It can be opened for extension (paralyse at the same time if it is at water area, it could get electric shocks IF we were to implement it) as well as closed for modification (has no effect on other classes operations). Therefore, it could be even more creative and simpler to modify and improve the game experience.

Submitted Google Form based on Creative Requirement 5 Questions:

1. What is the title of the requirement?
    a. New Weapon with status effect
    b. Lightning Blade, paralyze status effect

2. What is the summary of the feature?
    a. Paralyze effect could help players to run away from enemy (enemy with paralyze will not move until it overcomes the effect by a certain probability) and is especially effective towards Fish enemies

3. How and why does your proposed requirement adhere to the SOLID principles?
    a. The paralyze effect is created as an individual class and it handles only paralyze action that determines when the enemy can break through. Thus, the principle used in this requirement is S, single responsibility principle as it handles only paralyze action.
    b. Other than that, it also applies the Open Close principle as it will not affect other classes. It can be opened for extension (paralyze at the same time if it is at water area, it could get electric shocks IF we were to implement it) as well as closed for modification (has no effect on other classes operations).

4. The new requirement must use at least two classes from the engine package. Which classes are you going to use, and how will you use them in the design & implementation?
    a. The classes that are used by the new requirement from engine package are weapon, weaponItem, actor and action classes.
    b. To use it for the proper way of design and implementation:
        - We would let Lightning Blade extend WeaponItem.

- We would let Paralyze Action extend from Action classes.

c. This is due to the reasoning that we would have to execute the paralyze action to determine if the enemy is finally movable at each round. Thus, this will be called by the enemy (which it will extend from it's actor class) to call the action class. Finally, we would also create string output (menuDescription) to allow the player to know why the enemy is currently not moving!

5. The new requirement must use/re-use at least one (1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3). Which feature(s) are you going to use, and how will you use them in the design & implementation?
    a. - The existing feature used is the special skill that we have implemented in assignment 2. This is because that Lightning Blade weapon has a special skill (lightning shock) that can be chosen by the player to perform the paralyze effect onto a single target enemy.

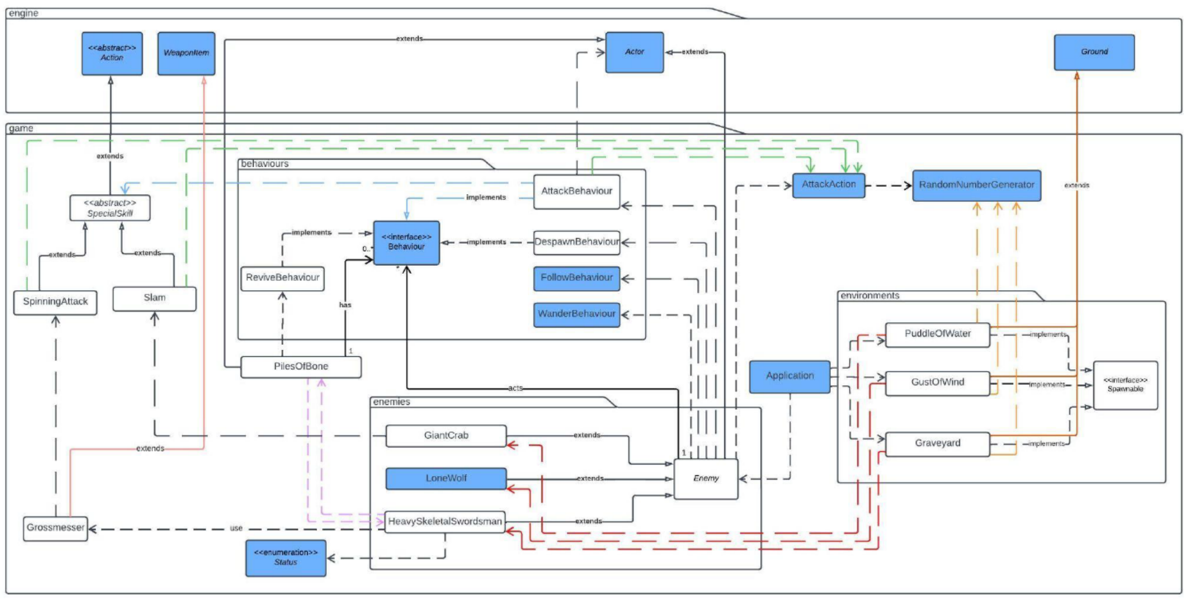
6. The new requirement must use existing or create new abstractions (e.g. abstract classes or interfaces, apart from the engine code). Which abstractions are you going to create/re-use, and how will you use them in the design & implementation?
    a. The abstract classes involved here are Weapon Item, Weapon and Action classes. As we design it this way , we can ensure Lightning Blade and Action would have less code repetition as it is a weapon item that can be used by a player to perform a special skill that will deal an AttackAction() and ParalyzeAction() onto the enemy.

7. The new requirement must use existing or create new capabilities. Which capabilities are you going to create/re-use, and how will you use them in the implementation?
    a. We will be reusing an enumeration class (Status) as we have previously used. Within here, we have added Paralyzed so that before an enemy performs any action, it will check if it is paralyzed or not. However a paralyzed enemy can never attack / move / wander / follow, until it has removed this status (it can move in the next round).
    b. With the reasoning here, we added it into enumeration so that we could add it into status of the enemy so we can reduce the string of literals used.
    c. For example, it will be space and cost wasting if we define a Boolean attribute to see if it has been paralyzed.

Changes and Improvements of Assignment 2:

- Every requirement listed below this section is assignment 1 and 2.
- Latest update on UML and rationale are labelled with a blue title after receiving feedback from assignment 2.

Requirement 1



Rationale:

PuddleOfWater, GustOfWind, Graveyard

- These classes extend the Ground abstract class in the engine package because the environment is ground.
- Since they use the same attribute and method, we inherit the ground class to avoid repetition (DRY).
- Three classes have a dependency on RandomNumberGenerator because they use RandomNumberGenerator to calculate the chance to spawn an enemy.

Spawnable

- An interface that is used to separate ground that can spawn enemies and those that can't. This interface makes sure that other environments don't have unnecessary tasks (LSP).

PuddleOfWater and GiantCrab

- PuddleOfWater has a dependency on GiantCrab because PuddleOfWater can spawn GiantCrab.

GustOfWind and LoneWolf

- GustOfWind have a dependency on LoneWolf because GustOfWind can spawn LoneWolf

Graveyard and HeavySkeletalSwordsman

- Graveyard has a dependency on HeavySkeletalSwordsman because Graveyard can spawn HeavySkeletalSwordsman.

Environment package

- PuddleOfWater, GustOfWind, Graveyard, and Spawnable are placed under the environment package. This is to organise similar classes as a unique package.

Enemies

- Enemy abstract class will extend the Actor abstract class as it is an actor and shares common attributes with the actor abstract class.
- All three enemies (GaintCrab, LoneWolf, HeavySkeletalSwordsman) will extend the abstract Enemy class. This is because they have similar attributes and methods, which could avoid repetition (DRY).
- Four of the classes mentioned above are placed under the Enemies package. This is to organise similar classes as a unique package to avoid confusion and helps to protect access capability.
- Enemy abstract class also has dependencies on several behaviours as they will call these behaviours' methods to perform interaction in game.

HeavySkeletalSwordsman

- It has a dependency on Grossmesser class because it has Grossmesser as a weapon.

PilesOfBone and HeavySkeletalSwordsman

- Both have a dependency on each other. This is because Swordsman has a unique ability that is changing itself into PileOfBones after being defeated, and PileOfBones will be revived with full health after 3 rounds of not being hit.

Grossmesser

- Grossmesser class will extend the WeaponItem abstract class because it is a weapon item and shares the common attributes and methods in the class. This avoids repetition(DRY).

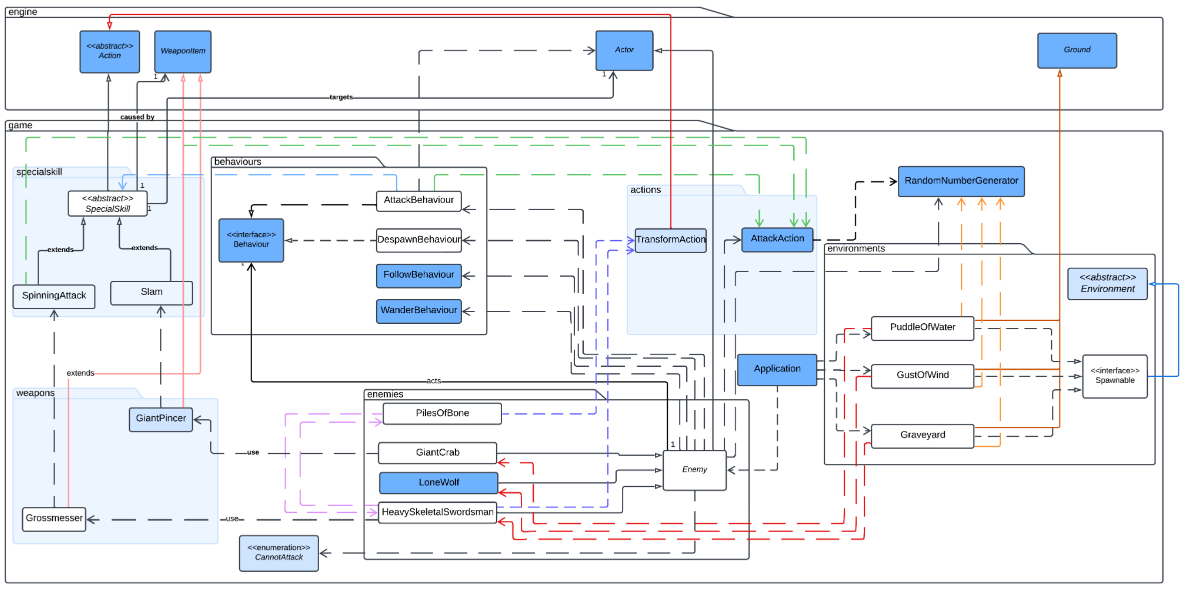
Behaviour

- Behaviour interface class has been provided, it allows behaviours to implement the same method. The rule followed here is the single responsibility principle as we want to ensure every new behaviour such as AttackBehaviour, DespawnBehaviour, and ReviveBehaviour will only perform a particular method when called.

SpecialSkill

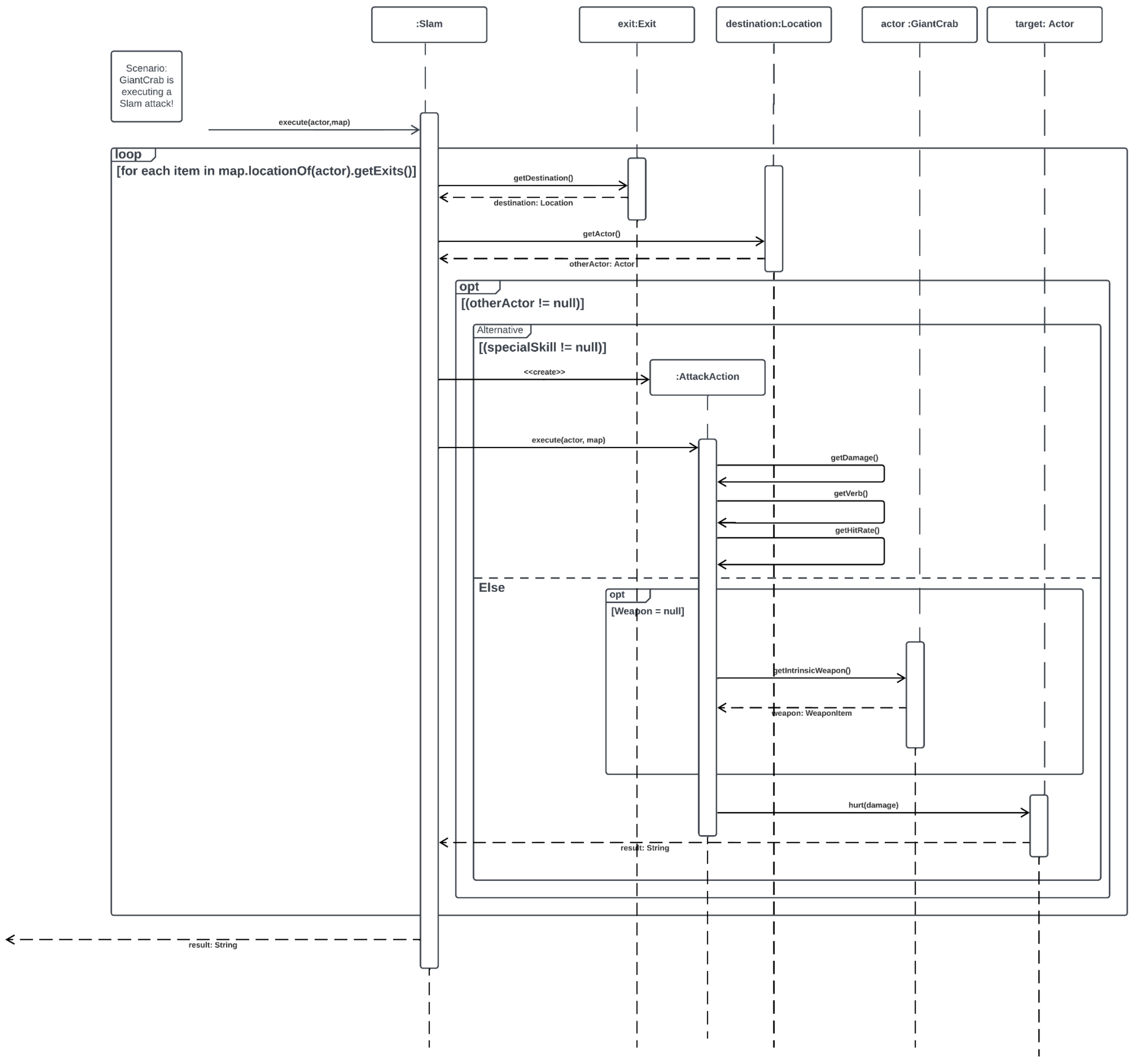
- SpecialSkill abstract class will extend the Action abstract class as it also shares common attributes and methods. This can help us to reduce some repetition code in SpecialSkill abstract class (DRY).
- The concept here is that we want to create a SpecialSkill class for special attack types such as Area Of Effect that can be performed by enemies.
- AttackBehaviour class will have dependencies on SpecialSkill class as some enemies may have SpecialSkill that can be performed.

Changes in Assignment 2 for Requirement 1



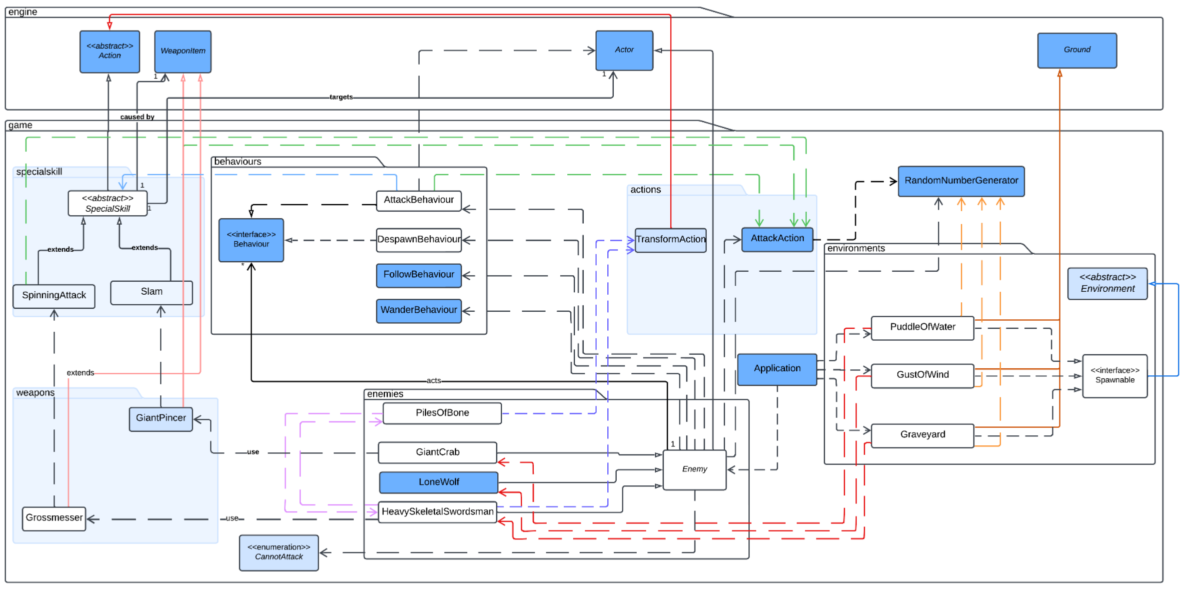
- The CannotAttack enumeration class is added to prevent the usage of literals. It is used to set the type that the enemy cannot attack to prevent them from attacking its own type.
- Weapon package is added as we are adding an additional weapon, GiantPincer which is usable by GiantCrab. Since now there are two weapons, such a weapons package is good for us to organise weapons as we have multiple weapons.
- The cons of doing enemy abstract classes might increase complexity as there are now more layers of abstraction classes. It might become a problem as we might miss out on some important methods to override which will cause issues to our program.

<u>Interaction Diagram for Requirement 1</u>



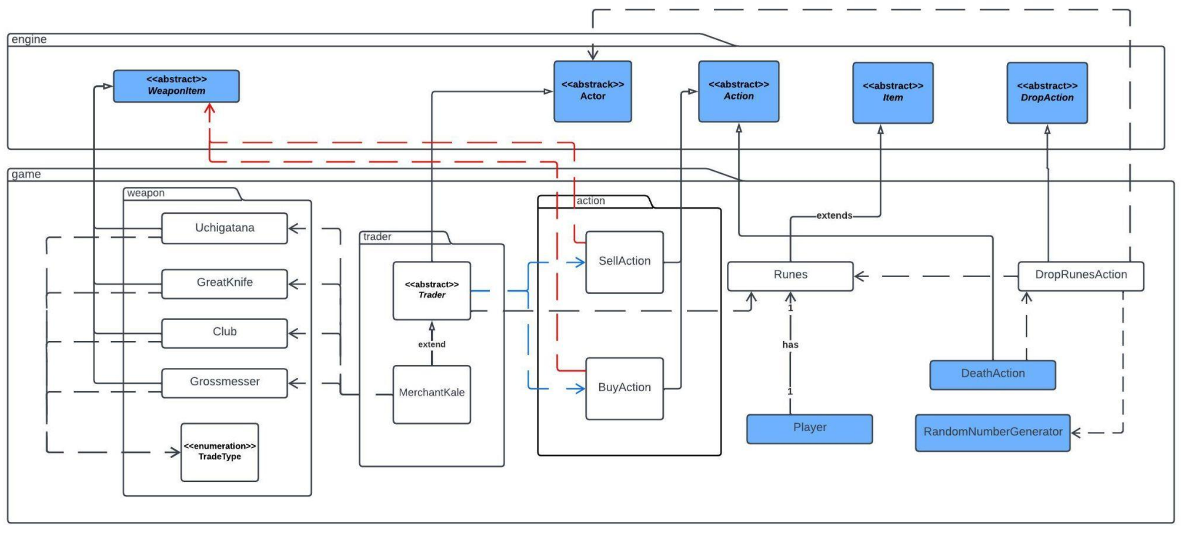
This sequence diagram shows a scenario where the GiantCrab performing a Slam area attack.

Changes in Assignment 2 for Requirement 1 (Latest feedback of assignment 2)



- All three enemies (GaintCrab, LoneWolf, HeavySkeletalSwordsman) will extend the abstract Enemy class. This is because they have similar attributes and methods, which could avoid repetition (DRY).
- It also achieves Liskov Substituition Principle as enemy type is the parent class that every subclass could be an type of enemy.
- The cons of doing enemy abstract classes might increase complexity as there are now more layers of abstraction classes. It might become a problem as we might miss out on some important methods to override which will cause issues to our program.
- SpecialSkill abstract class will extend the Action abstract class as it also shares common attributes and methods. This can help us to reduce some repetition code (DRY).
- Alternative way for SpecialSkill could be implementing different interfaces as some specialSkill may be Area Of Effect, Single Target. Thus, it achieves interface segregation principle (ISP) as we could classify specialSkill to use the suitable interface classes.
- Behaviour interface class allows behaviours to implement the same method. The rule followed here is the single responsibility principle as we want to ensure every new behaviour will only focus on its own jobs.

Requirement 2



Rationale:

Trader

- Trader extends the Actor abstract class because Trader is an actor, since they have the same basic attributes and method, we use an abstract class to avoid repetition(DRY).
- It will also be an abstract class to allow the extension of the game by having more traders. So to reduce repetition (DRY) we make traders an abstract class.
- Trader has a dependency on BuyAction and SellAction because Traders provide buy and sell action to players to buy or sell weapons.
- Trader has a dependency on Runes because traders need to use runes to perform sell and buy weapon action.

MerchantKale

- MerchantKale extends Trader abstract class because MerchanrtKale is a Trader, since they have the same basic attributes and method, we can use an abstract class to prevent repetition(DRY).
- MerchantKale has dependencies on Uchigatana, GreatKnife, Club, Grossmesser because MerchantKale can sell Uchigatana, GreatKnife, Club to the player and buy Uchigatana, GreatKnife, Club, Grossmesser from the player.

Weapon

- Uchigatana, GreatKnife, Club, and Grossmesser are all WeaponItem, so they extend WeaponItem abstract class to prevent repetition(DRY).
- Uchigatana, GreatKnife, Club, and Grossmesser have a dependency TradeType enumeration because some weapons can only be sold or be sold and bought. So we use TradeType to categorise them(AVOID EXCESSIVE USE OF LITERALS).
- Uchigatana, GreatKnife, Club, Grossmesser and TradeType are placed under the weapon package. This is to organise similar classes as a unique package to avoid and helps to protect access capability.

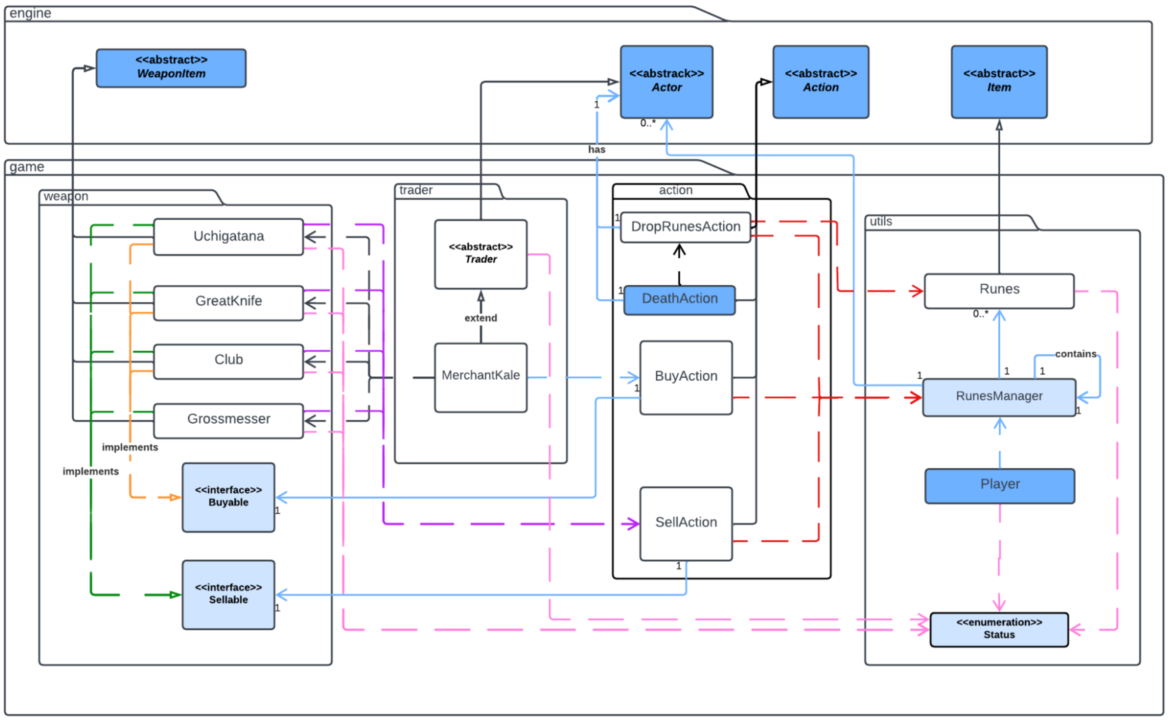
Action

- SellAction and BuyAction both extend the Action abstract class because they are all action. So we use an abstract class to prevent repetition(DRY).
- SellAction and BuyAction both have a dependency on WeaponItem because they are responsible for the action to sell weapons and buy weapons.
- SellAction and BuyAction are placed under the action package. This is to organise similar classes as a unique package to avoid and helps to protect access capability.

Runes:

- It extends the item abstract class because it behaves similarly to common attributes and methods. This could avoid much repetition code (DRY).
- DeathAction would also lead to DropRunesAction as runes will be dropped after the player is dead, and such point runes that extend the item can be placed as a symbol on the map that can be further picked up by the player for a second chance.
- DropRunesAction will extend DropItemAction since Runes are declared as items for our design concept. Thus, this could also reduce code repetition (DRY).

<u>Changes in Assignment 2 for Requirement 2</u>



RunesManager

- Following Single Responsibility Principle(SRP), is a good reason to have a RunesManager to handle all runes transaction in the game rather than handle by different class.

Status

- Status is an enumeration that use by WeaponItem, actor and items.
- The reason of using enumeration is to prevent overload usage of literals. We can use Status enumeration to catograrise each item or actor.

Buyable

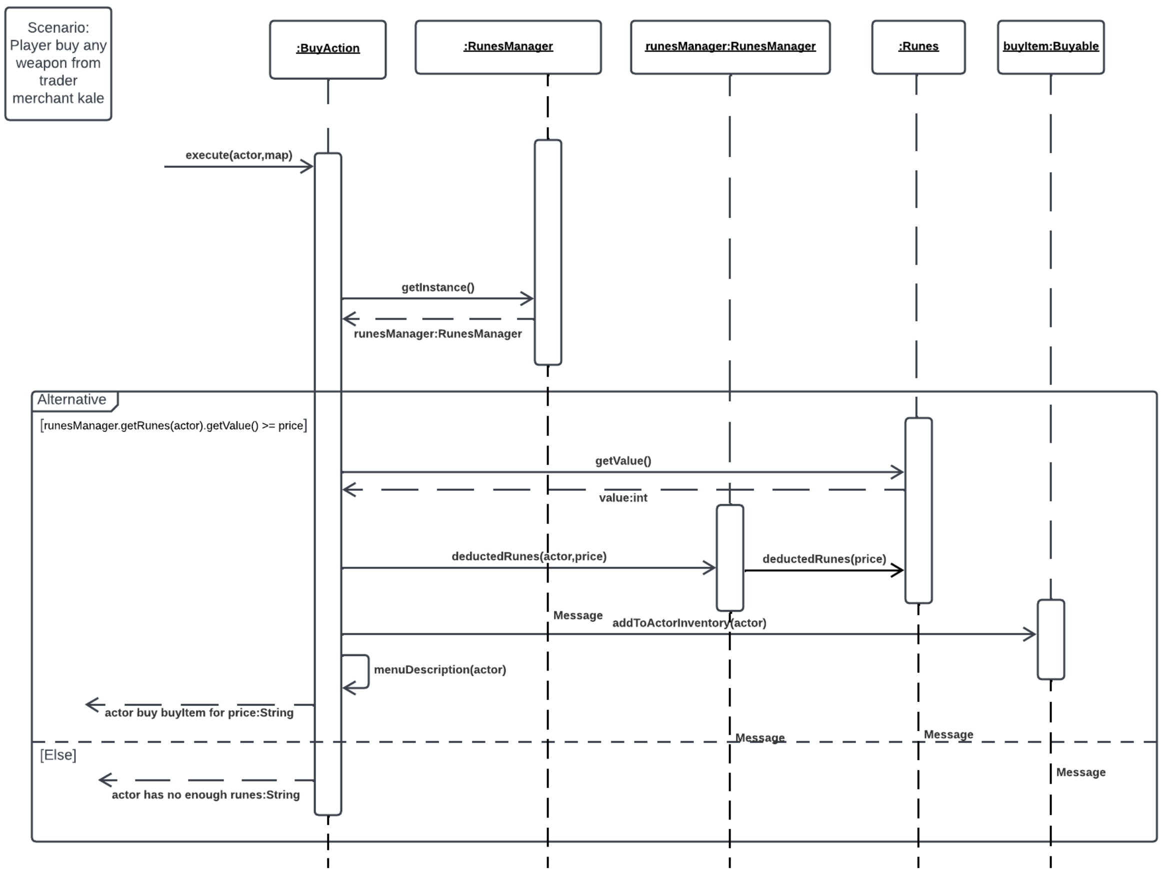
- Buyable is a interface and can be implement by WeaponItem such as Uchigatana, GreatKnife, Club

- We use interface because of interface segregation principle(LSP) because not every WeaponItem can be buy.

Sellable

- Sellable is an interface and can be implemented by WeaponItem such as Uchigatana, GreatKnife, Club, Grossmeser
- We use interface because of interface segregation principle(LSP) because not every WeaponItem can be sold.

Interaction diagram for Requirement 2

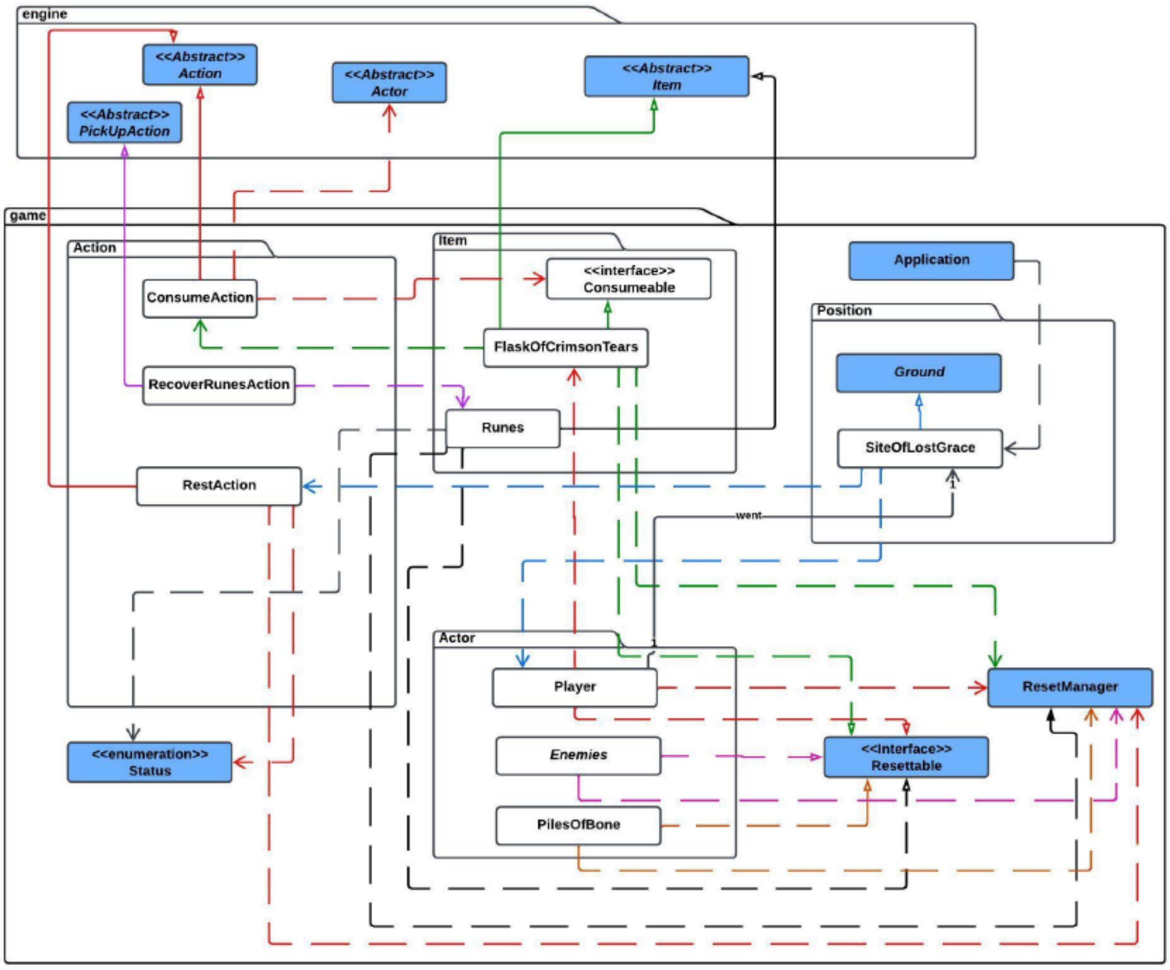

Scenario: Player buy any weapon from trader merchant kale
:BuyAction
:RunesManager
runesManager:RunesManager
:Runes
buyItem:Buyable
execute(actor,map)
getInstance()
runesManager:RunesManager
Alternative
[runesManager.getRunes(actor).getValue() >= price]
getValue()
value:int
deductedRunes(actor,price)
deductedRunes(price)
Message
addToActorInventory(actor)
menuDescription(actor)
actor buy buyItem for price:String
Message
Message
[Else]
Message
actor has no enough runes:String

This sequence diagram shows a scenario where the player buy weapon from trader merchant kale.

Changes in Assignment 2 for Requirement 2 (Latest feedback of assignment 2)

- Player have status PLAYER_DIED to check if the player is dead, if is dead it will perform deadaction.

Requirement 3



Rationale:

Flask of Crimson Tears

- It inherits the item as it is an item to follow don't repeat yourself principle. The benefit is we do not have to repeat the attributes and methods that an item already has.
- It has a dependency on ConsumeAction because a player can consume it.
- It implemented the interface consumable because it can be consumed. The benefit is to allow other classes to use the method that a consumable has.

ConsumeAction

- It is created because consuming is a single action, so it follows the single responsibility principle.
- It inherits action as it is an action, this is to follow the don't repeat yourself principle.
- It has a dependency on consumable because it allows actors to consume a consumable. This has the benefit of allowing it to access methods that a consumable has.
- The benefit of implementing this is that the player will be able to choose to consume the item, and it will be easy to extend as consumables can all return this ConsumeAction for the player to choose to consume it.

Consumable

- It is created for items that can be consumed to implement and prevent multi-level inheritance. It ensures that classes that implement it can be consumed.
- The benefit is that it makes the game easier to extend as all items that can be consumed only need to implement this interface.

Site of Lost Grace

- It is a unique ground, hence it inherits ground to follow don't repeat yourself principle.
- It allows the player to rest on it, so it has dependencies on RestAction and Player.

RestAction

- It inherits action as it is an action to follow don't repeat yourself principle.
- It has a dependency on ResetManager because the game will reset when the player is in rest action
- It has a dependency on Status because it will put the player in rest status.
- The benefit is that all actors that can use allowableAction will be provided with an option to rest on it, so it can be easily extended

Game Reset

- ResetManager class only has one instance of itself and cannot be created externally, this ensures that there is only one instance throughout the game.
- All enemies are resettable, hence the Enemy abstract class implements the resettable interface.
- The player is resettable, hence the Player class implements the resettable interface.
- FlaskOfCrimsonTears is resettable, hence the FlaskOfCrimsonTears class implements the resettable interface.
- All of them have a dependency on ResetManager because they will register themselves as resettable in the ResetManager
- Players will respawn in the last site of lost grace they visited, so it will save the last site of lost grace. Hence, there is an association between Player and SiteOfLostGrace.
- It can be easily extended as all resettable can just implement the resettable interface and register themselves as resettable to be involved in the reset process.

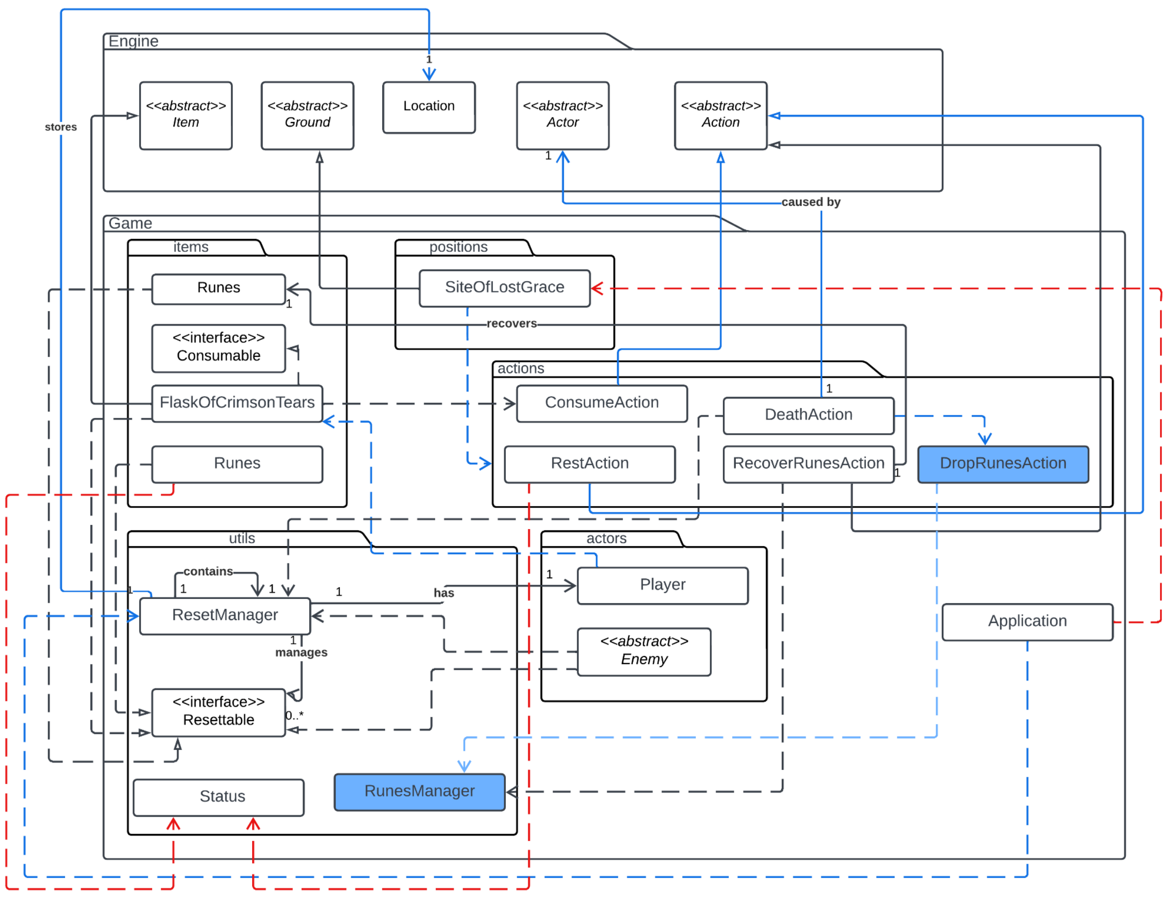
RecoverRunesAction

- It extends the action class as it is an action, this could prevent the repetition of code (DRY).
- It has a dependency on runes because it needs to know which runes to pick up.

Runes

- It has a dependency on status because it will need to check if the player is resting to determine whether it should disappear. It also implements resettable as it is resettable when dropped.
- The cons that it implements resettable is that it only resets when the player dies, hence it is not a resettable when a player is alive.

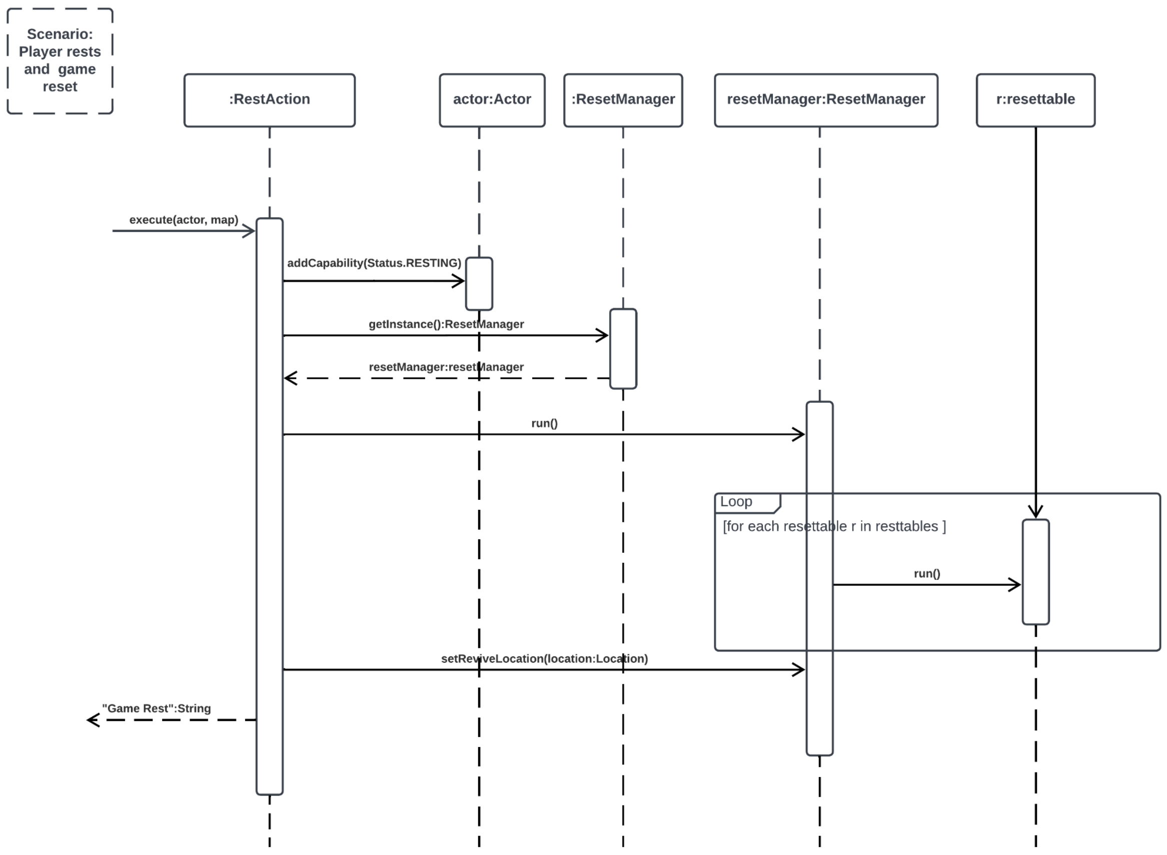
<u>Changes in Assignment 2 for Requirement 3</u>



Changes in Assignment 2:

- RecoverRunesAction is updated to extend the action class instead of pickupAction as it can only recover runes items. The benefit of this is it follows the single responsibility principle as it is only responsible for recovering runes. The drawback is that it might be better if we inherit pickupAction so that we follow the DRY principle.
- Added a dependency from the application to ResetManager to set the revive location of the player. The drawback is the location is hard coded, but it can be easily extended by changing the location input.
- Updated to use RunesManager in runes recovering and dropping. The pros is that is makes the management of runes easier throughout the game.
- Updated to use DropRunesAction in death action to manage dead actor runes to follow SRP
- PilesOfBone is changed to inherit enemy abstract class as it is an enemy. The cons that PilesOfBone inherits enemy is that it does not have most of the behaviour that an enemy has. But the pros is that it has most of the attributes that enemy has.

Interaction Diagram for Requirement 3



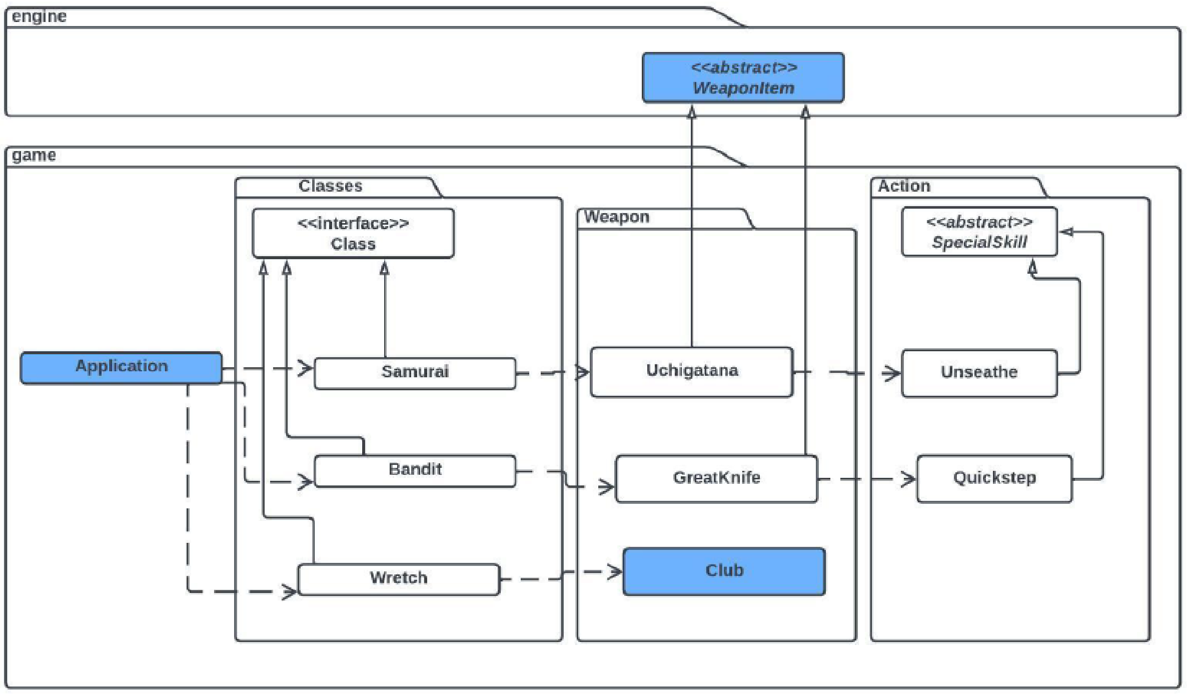
Scenario: Player rests and game reset
:RestAction
actor:Actor
:ResetManager
resetManager:ResetManager
r:resettable
execute(actor, map)
addCapability(Status.RESTING)
getInstance():ResetManager
resetManager:resetManager
run()
Loop
[for each resettable r in resttables]
run()
setReviveLocation(location:Location)
"Game Rest":String

This sequence diagram shows a scenario where the player rests at site of lost grace and the game resets.

Changes in Assignment 2 for Requirement 3 (Latest feedback of assignment 2)

- No changes

Requirement 4



Rationale:

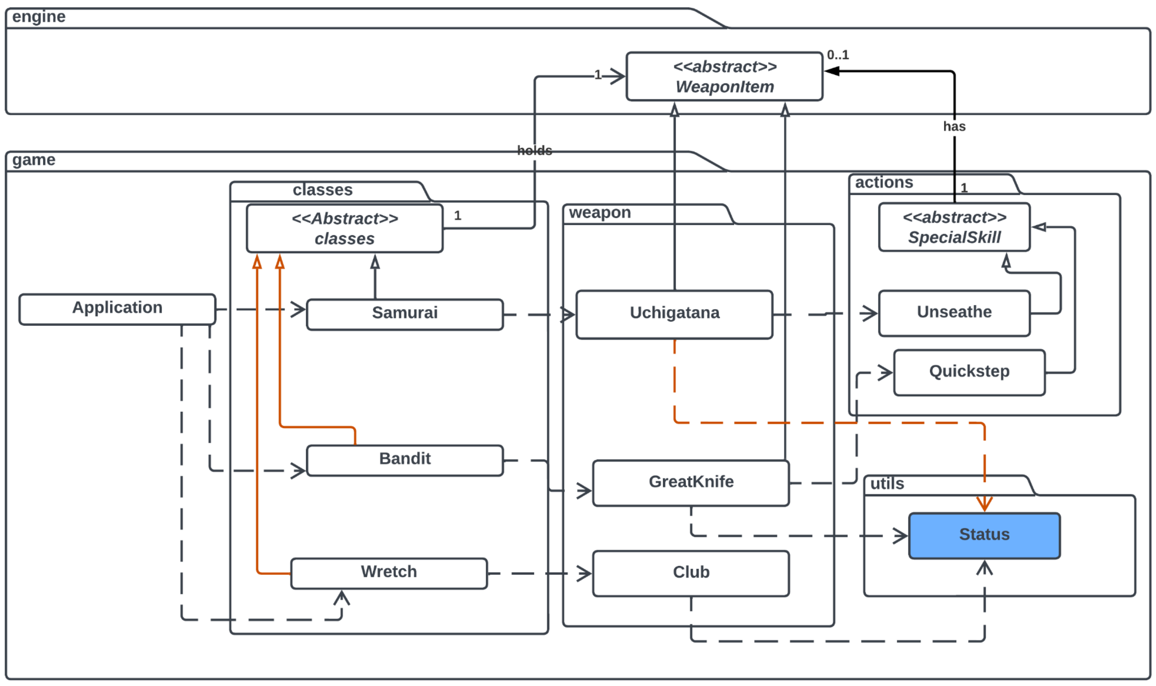
Classes and Combat Archetypes

- Before the game starts, the Application allows the player to choose one of the three classes, Application class has dependencies on Samurai, Bandit and Wretch.
- Class is an abstract class for classes that players can choose. This is to follow don't repeat yourself principle as we don't need to repeat the methods and attributes for each class.
- Samurai, Bandit and Wretch are classes that player can choose from, hence they inherit and extends the Class class. This is to follow don't repeat yourself principle to not repeat similar attributes and methods.
- Samurai has a dependency on Uchigatana because it has Uchigatana as its starting weapon.
- Bandit has a dependency on GreatKnife because it has GreatKnife as its starting weapon.
- Wretch has a dependency on Club because it has Club as its starting weapon.
- The pros of implementing classes this way are that they can be extended easily when we want to add more classes. We can just extend the new class with the class classes.

Weapons

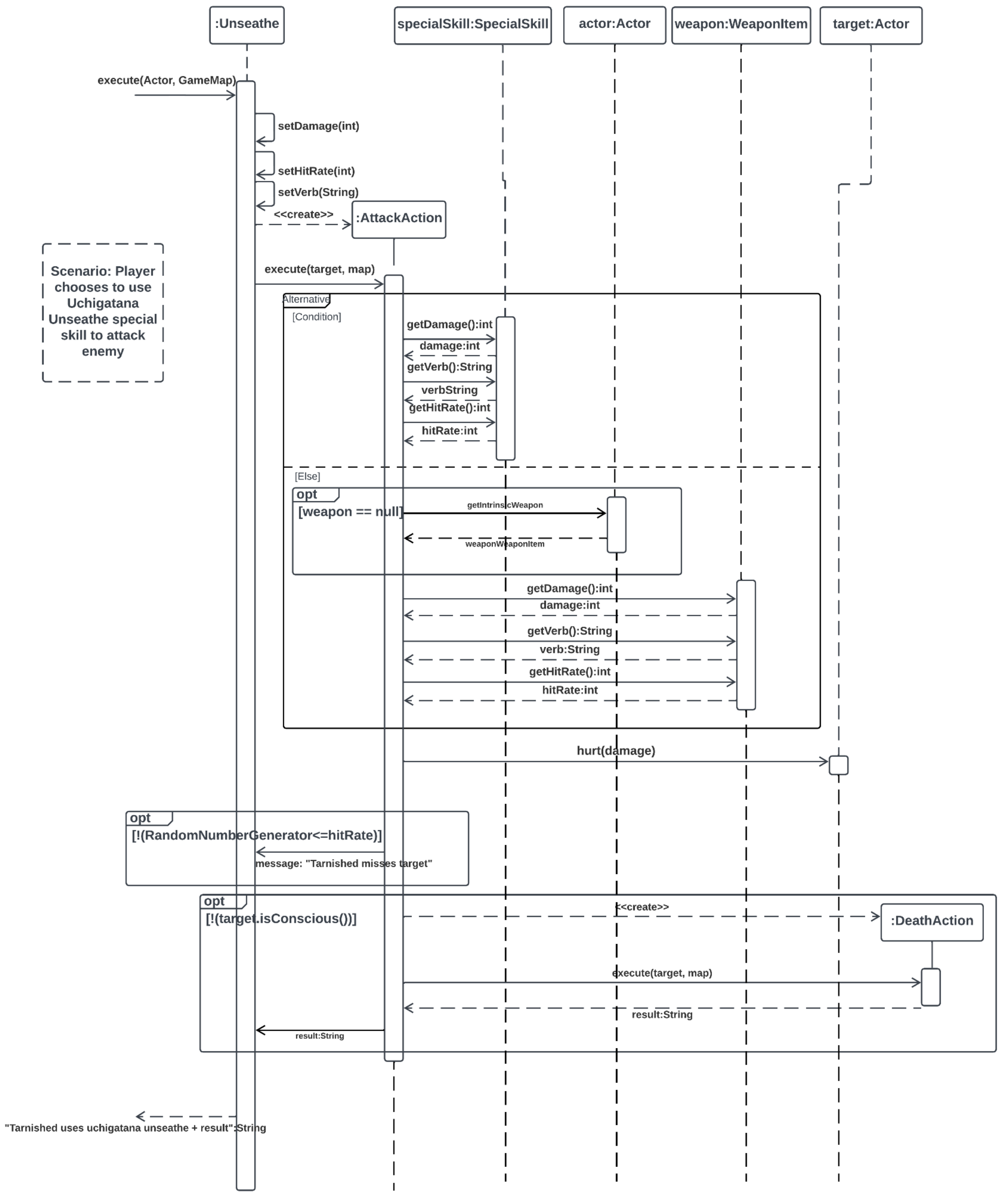
- Uchigatana and Great Knife are both WeaponItem, hence they inherit and extend the WeaponItem class to follow the don't repeat yourself principle.
- Uchigatana has a dependency on Unseathe because it has Unseathe as its special skill
- Great Knife has a dependency on QuickStep because it has QuickStep as its special skill
- Both Unseathe and QuickStep inherits and extends the SpecialSkill abstract class because they are both special skill. This follows the don't repeat yourself principle.

Changes in Assignment 2 for Requirement 4



- Dependencies from some weapons to status have been added to indicate whether the weapon has a special skill.
- An association from classes to weaponItem is added because each class has its own starting weapon. The pros it that it will be easy for the application to get the weapon and assign it to the player, the cons is the class can only have one weapon. To extend this in the future, the classes might have a list of weaponItems.
- An association is added from SpeicalSkill to WeaponItem as some SpecialSkill needs to calculate its damage by its weapon. The cons is some SpecialSkill does not rely on weapon so it is redundant.

Interaction Diagram for Requirement 4

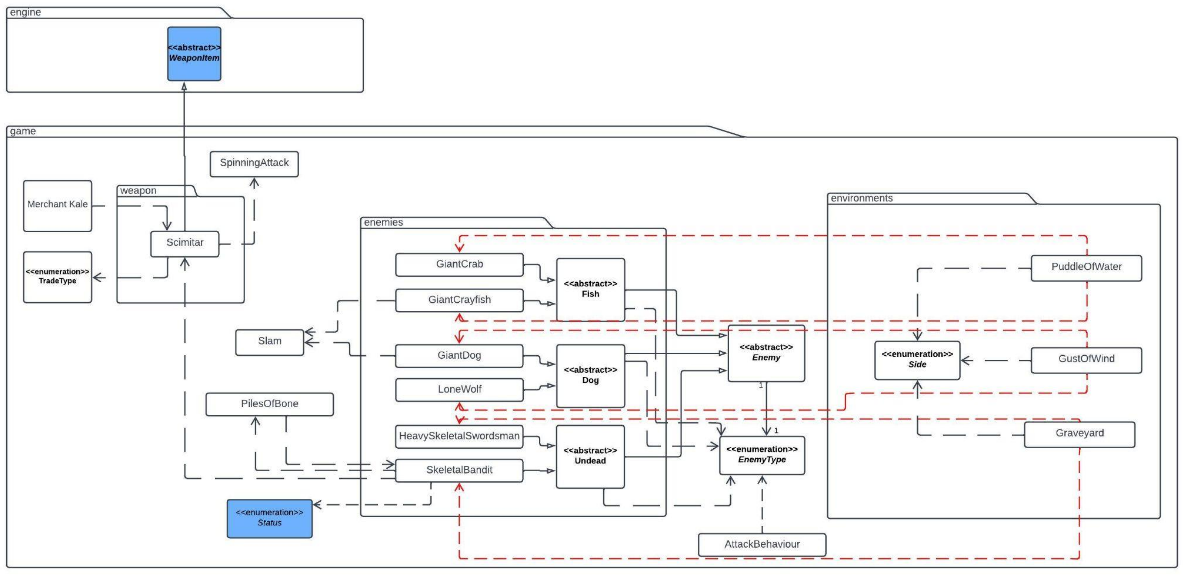


This sequence diagram shows the scenario where player chooses to use uchigatana special skill unsheathes on an enemy.

Changes in Assignment 2 for Requirement 4 (Latest feedback of assignment 2)

- No changes

<u>Requirement 5</u>



Rationale:

More enemies:

<u>Subclasses (abstract classes) of Enemy abstract class</u>

- Under this requirement, we have more enemies variations for Skeletons, Mammals and Sea creatures. A special relationship is that a specific enemy type should not attack its own kind or its variations.
- For example, GiantCrab and GiantCrayFish can be created by initialising Fish as every parent class can perform whatever the children class can perform, it will specify its type to avoid attacking each other.
- Several abstract subclasses that extend the enemy abstract class are created.
  - Abstract classes: Fish, Dog, Undead
  - Every subclass has a dependency on the enumeration EnemyType to set itself to one of the types.
  - For example, GiantCrab and GaintCrayfish will not attack each other as they have the same type.
  - Attack behaviour has a dependency on the enumeration EnemyType to check if the other actor has the same type.
- This concept is inspired by the Liskov Substitution Principle, it helps us to achieve polymorphism even if classes are abstract.
- This implementation allows us to achieve less repetition code (DRY) as we can define its type at the parent class.
- New enemies such as GiantCrayfish and GiantDog have dependencies onto Slam class as they can call the method to perform attack of Area Of Effect.
- However, the cons here is that there are more abstract class that extends one another. This could lead to complexity confusion in future.

<u>Scimitar</u>

- A Scimitar class will extend the WeaponItem abstract class because it shares the common attributes and methods in the class. This allows us to avoid repetition code (DRY).
- Scimitar has a dependency on TradeType enumeration because it only can be sold. So we use TradeType to categorise it(AVOID EXCESSIVE USE OF LITERALS).

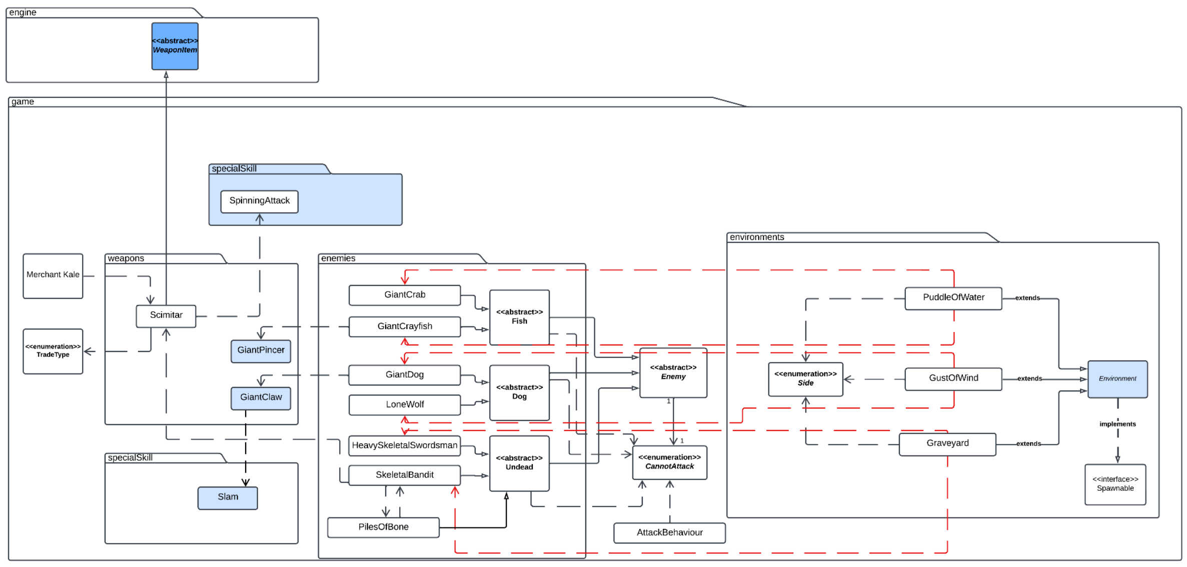
<u>MerchantKale</u>

- MerchantKale have a dependency on Scimitar because MerchantKale can sell Scimitar to the player and buy Scimitar from the player.

<u>PuddleOfWater, GustOfWind, Graveyard</u>

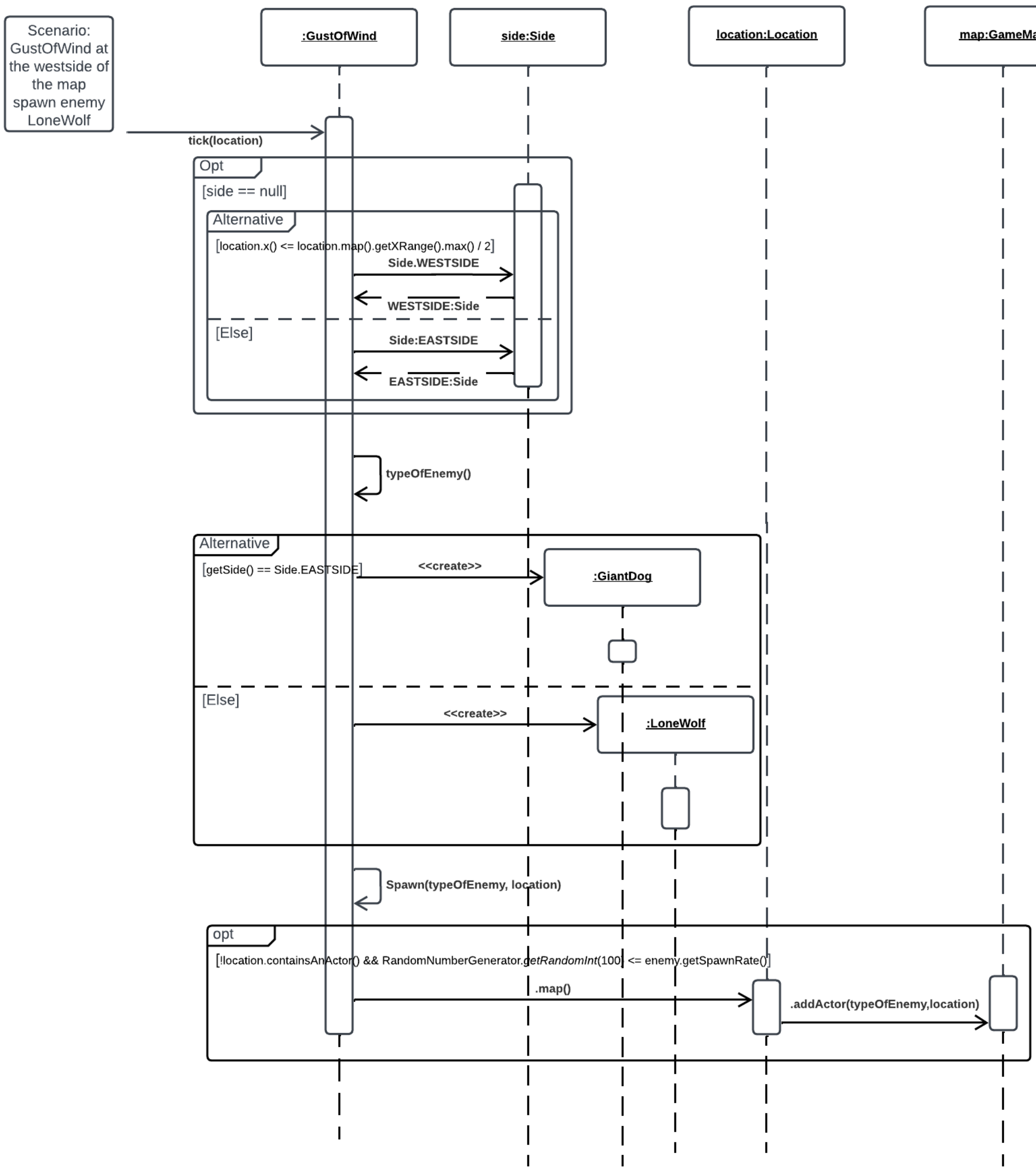
- PuddleOfWater has a dependency on GiantCrab and GiantCrayfish because PuddleOfWater will spawn GiantCrab if it is on the west side of the map, spawn GiantCrayfish if it is on the east side of the map.
- GustOfWind has a dependency on GiantDog and LoneWolf because GustOfWind will spawn LoneWolf if it is on the west side of the map, and spawn GianDog if it is on the east side of the map.
- Graveyard has a dependency on SkeletalBandit and HeavySkeletalSwordsman because Graveyard will spawn HeavySkeletalSwordsman if it is on the west side of the map, and spawn SkeletalBandit if it is on the east side of the map.
- PuddleOfWater, GustOfWind, and Graveyard have dependencies on Side enumeration because these environments can be on the west side or east side of the map. So we use Side to categorise them(AVOID EXCESSIVE USE OF LITERALS).

<u>Changes in Assignment 2 for Requirement 5</u>



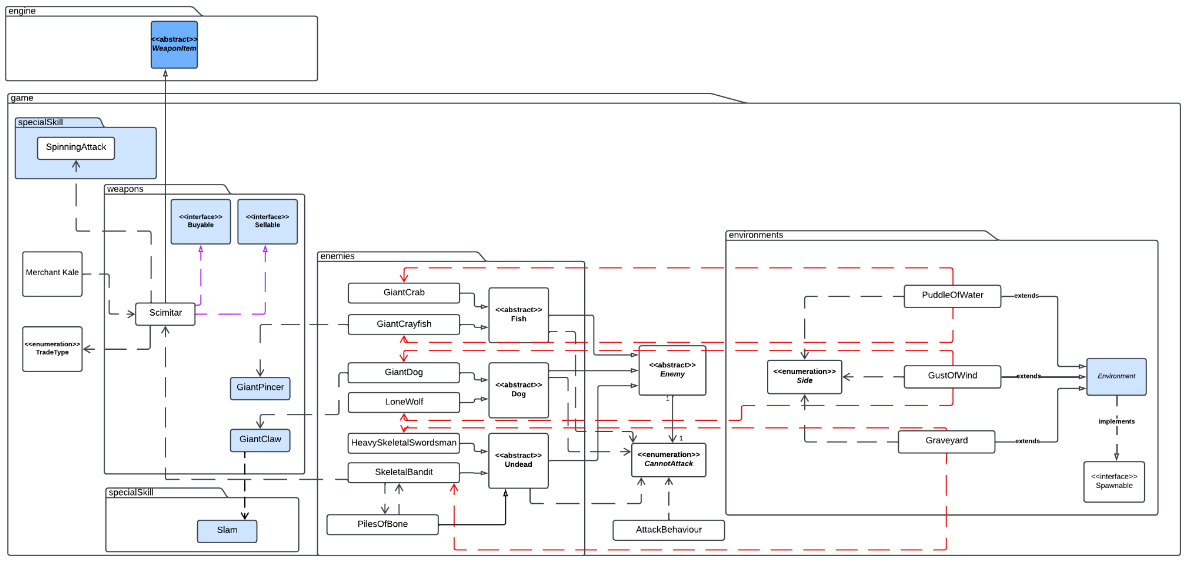
- Spawnable is now implemented by an abstract class, this is because we found out that environments share many common methods and attributes. Thus, this can achieve DRY by creating an abstract environment class to allow every subclass (environments) to extend it.
- Spawnable interfaces will be able to enforce every subclass of the environment parent class to implement the method within it (Thus, every environment can spawn enemy)
- specialSkill package is added to store spinning attacks within it. This is because we already have other special skills such as unseathe and slams. Thus, it will be more organised with a package specially designed for the special skill.

Interaction Diagram for Requirement 5


Scenario: GustOfWind at the westside of the map spawn enemy LoneWolf
:GustOfWind
side:Side
location:Location
map:GameMap
tick(location)
Opt
[side == null]
Alternative
[location.x() <= location.map().getXRange().max() / 2]
Side.WESTSIDE
WESTSIDE:Side
[Else]
Side:EASTSIDE
EASTSIDE:Side
typeOfEnemy()
Alternative
[getSide() == Side.EASTSIDE]
<<create>>
:GiantDog
[Else]
<<create>>
:LoneWolf
Spawn(typeOfEnemy, location)
opt
[!location.containsAnActor() && RandomNumberGenerator.getRandomInt(100) <= enemy.getSpawnRate()]
.map()
.addActor(typeOfEnemy,location)

This sequence diagram shows a scenario where GustOfWind at the westside of the map spawn enemy LoneWolf.

Changes in Assignment 2 for Requirement 5 (Latest feedback of assignment 2)



The only changes made here is to update the UML diagram. Scimitar weaponItem should implement both buyable and sellable interfaces as designed and implemented by us in the implementation of assignment 2.