AppliedSession21_Group8

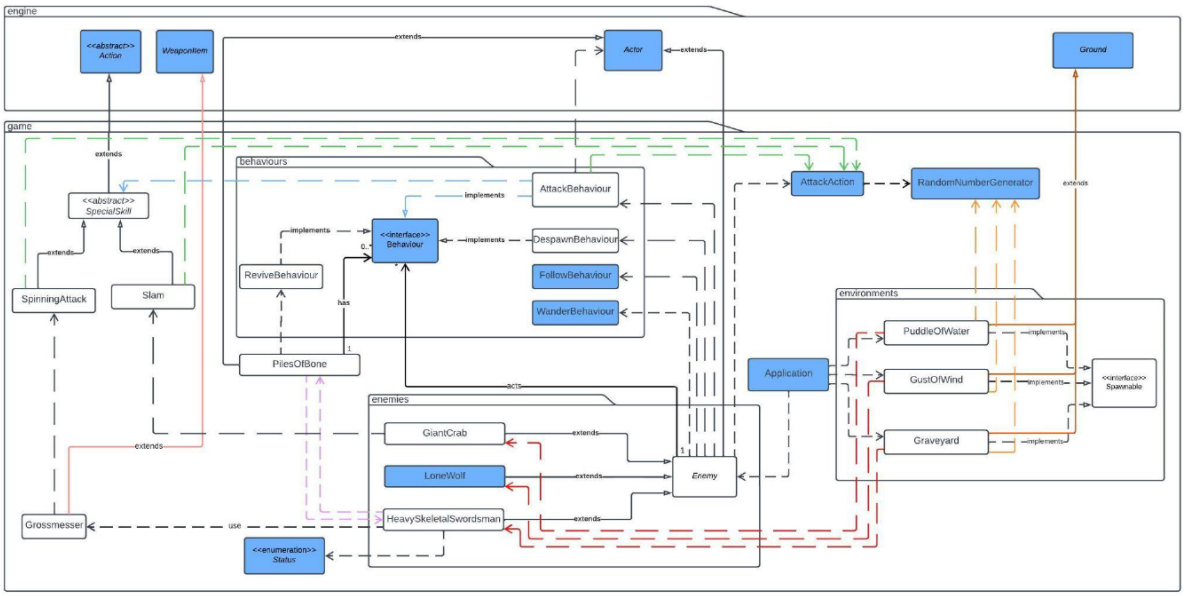
FIT2099 Assignment 2: Design

(modified by 3/5/2023)

Group member:

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

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).

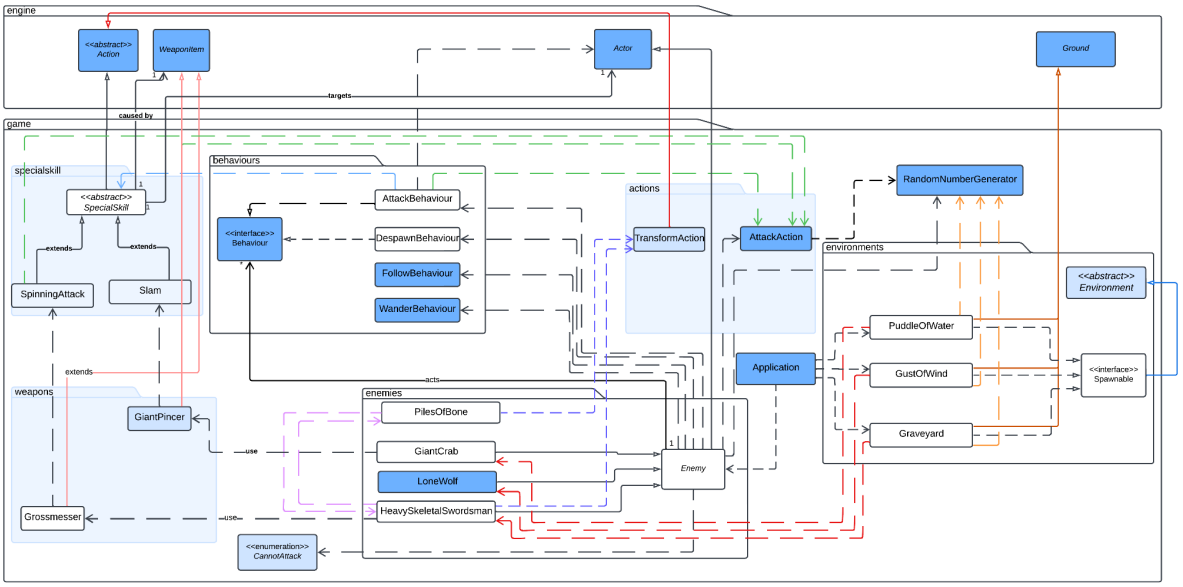
Behaviour

- 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

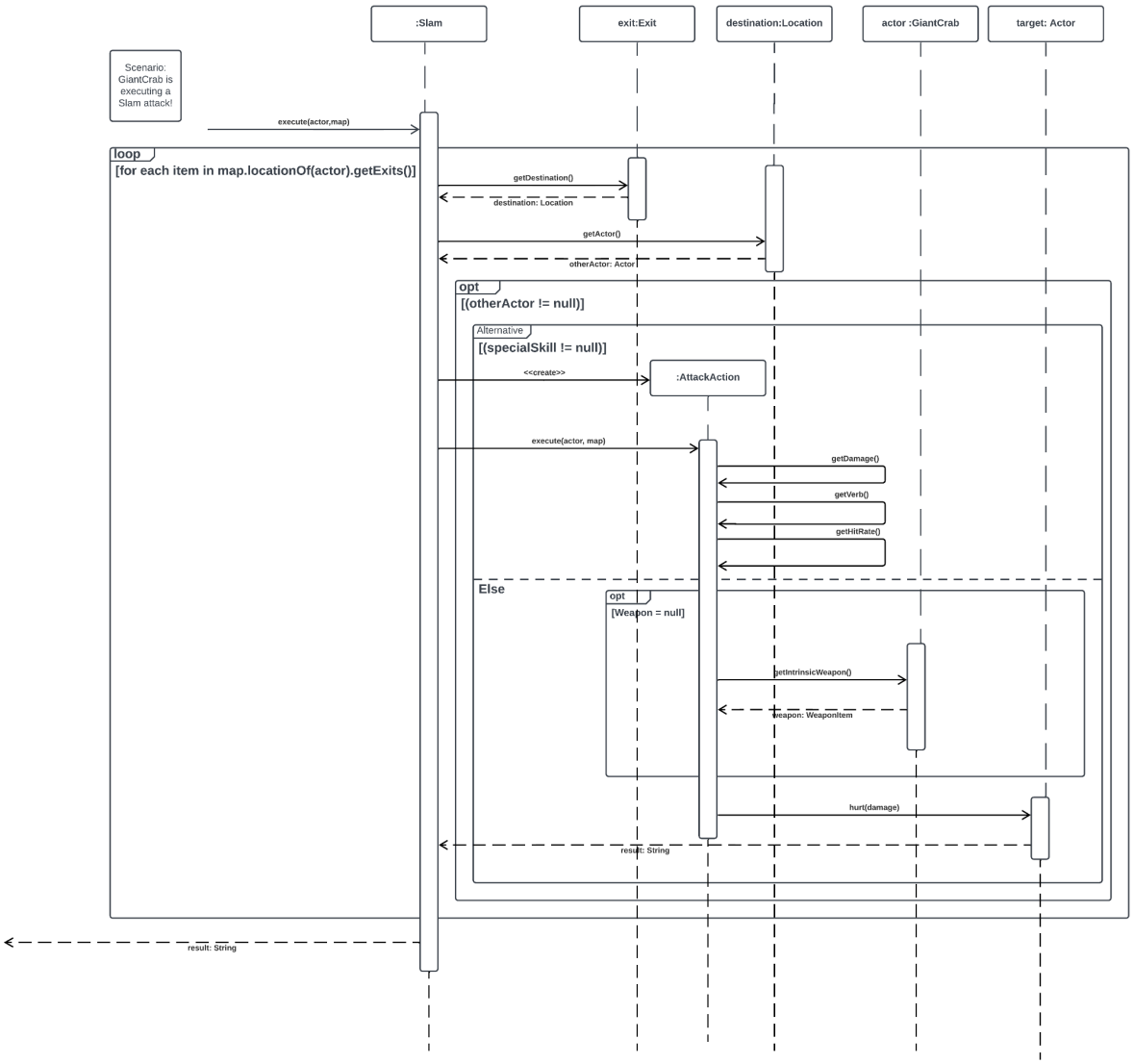
- SpecialSkill abstract class will extend the Action abstract class as it also shares common attributes and methods. This can help us to reduce some repetition code in SpecialSkill abstract class (DRY).
- The concept here is that we want to create a SpecialSkill class for special attack types such as 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



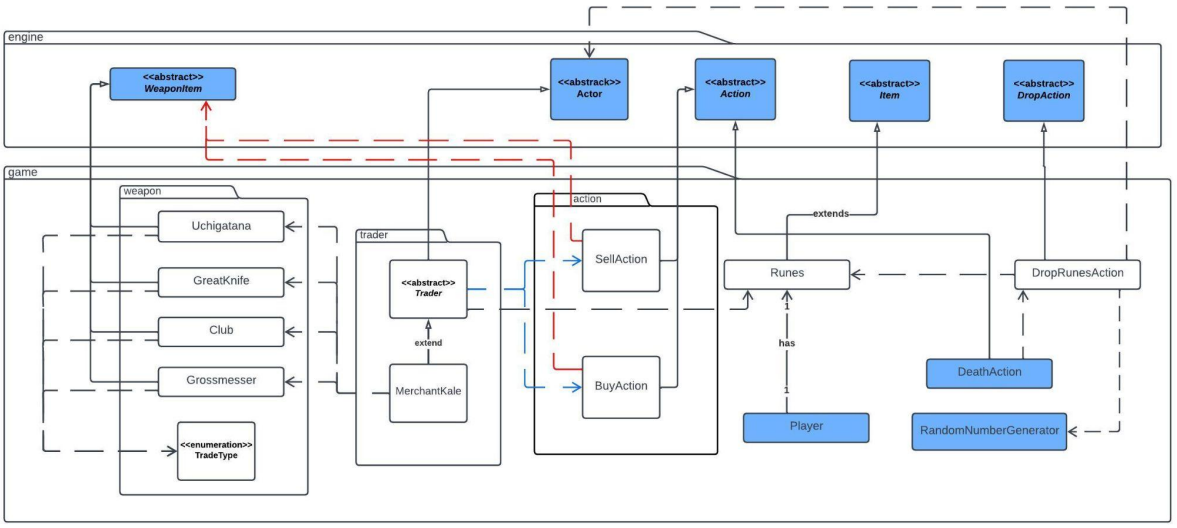
- The CannotAttack enumeration class is added to prevent the usage of literals. It is used to set the type that the enemy cannot attack to prevent them from attacking its own type.
- Weapon package is added as we are adding an additional weapon, GiantPincer which is usable by GiantCrab. Since now there 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.

Interaction Diagram for Requirement 1



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

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.

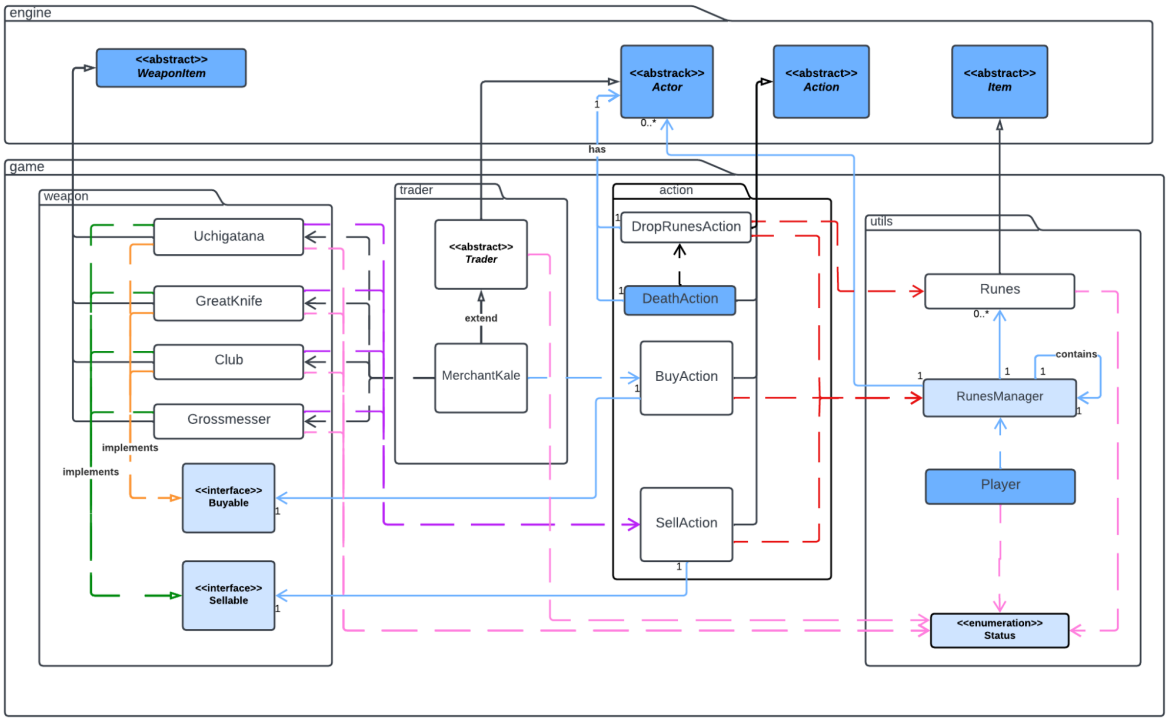
Action

- 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).

Changes in Assignment 2 for Requirement 2



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.

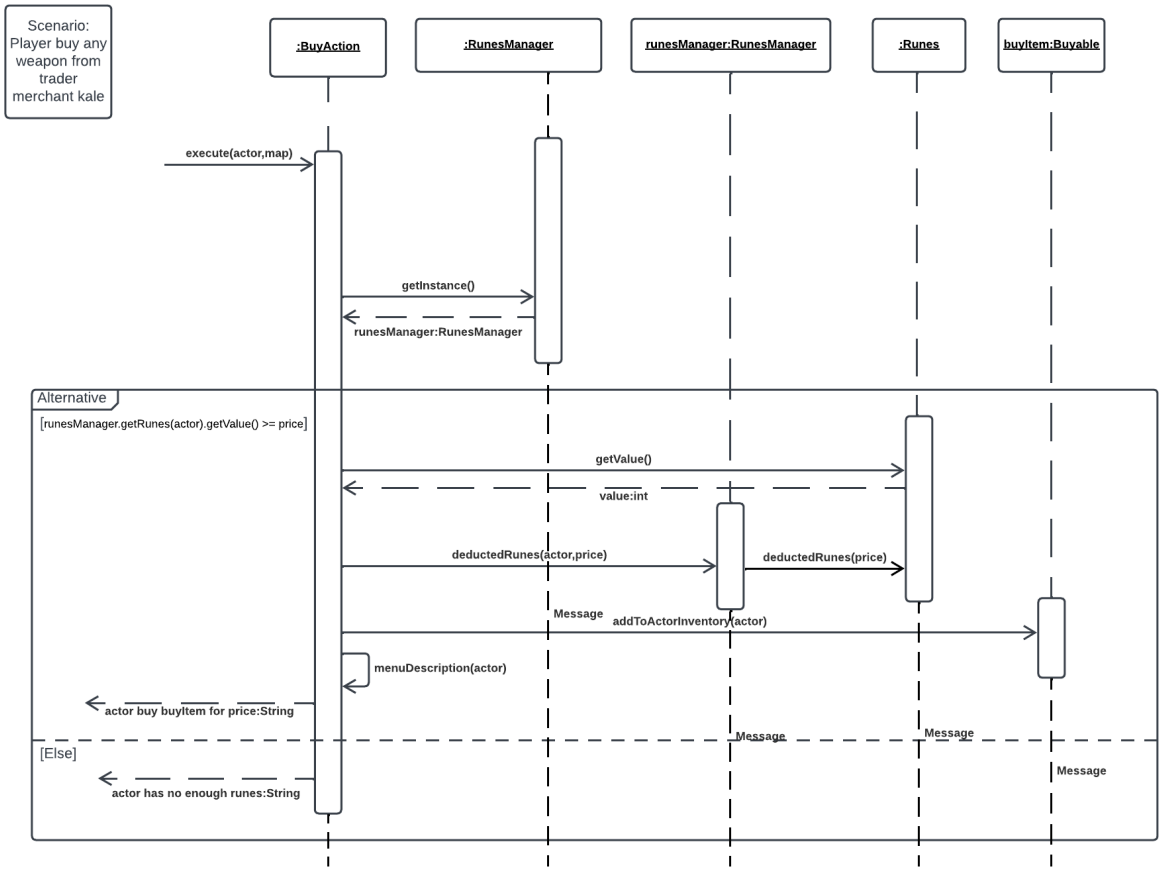
Buyable

- 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

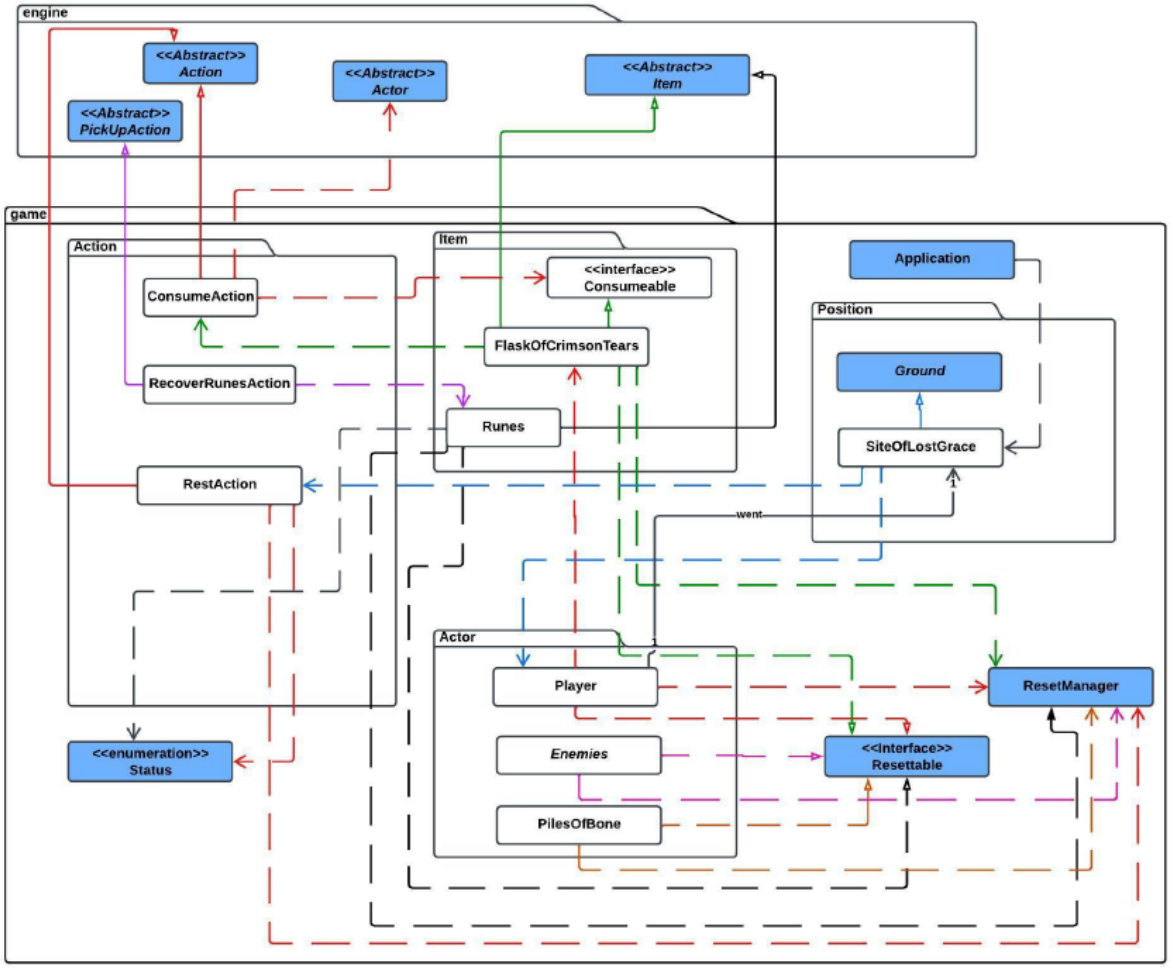
- Sellable is an interface and can be implemented by WeaponItem such as Uchigatana, GreatKnife, Club, Grossmeser
- We use interface because of interface segregation principle(LSP) becuase not every WeaponItem can be sell.

Interaction diagram for Requirement 2



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

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.

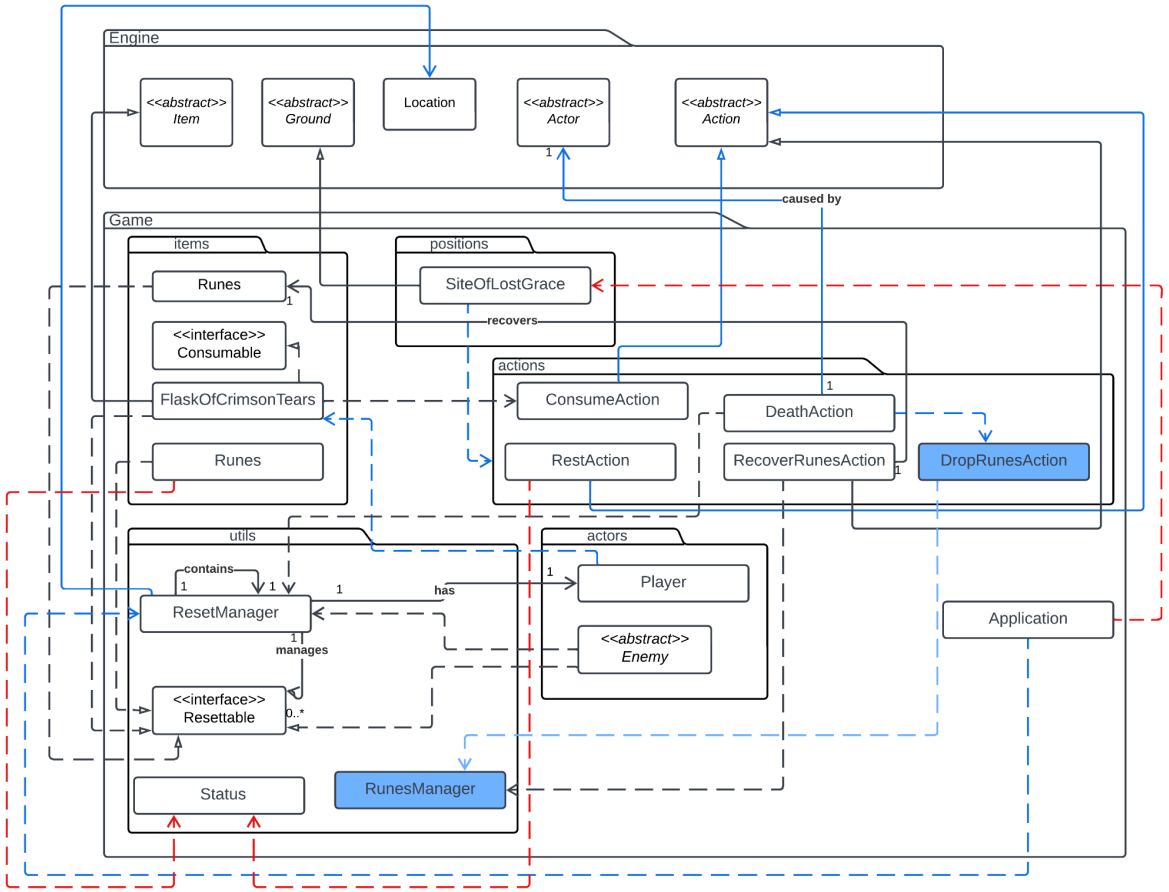
RecoverRunesAction

- 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.

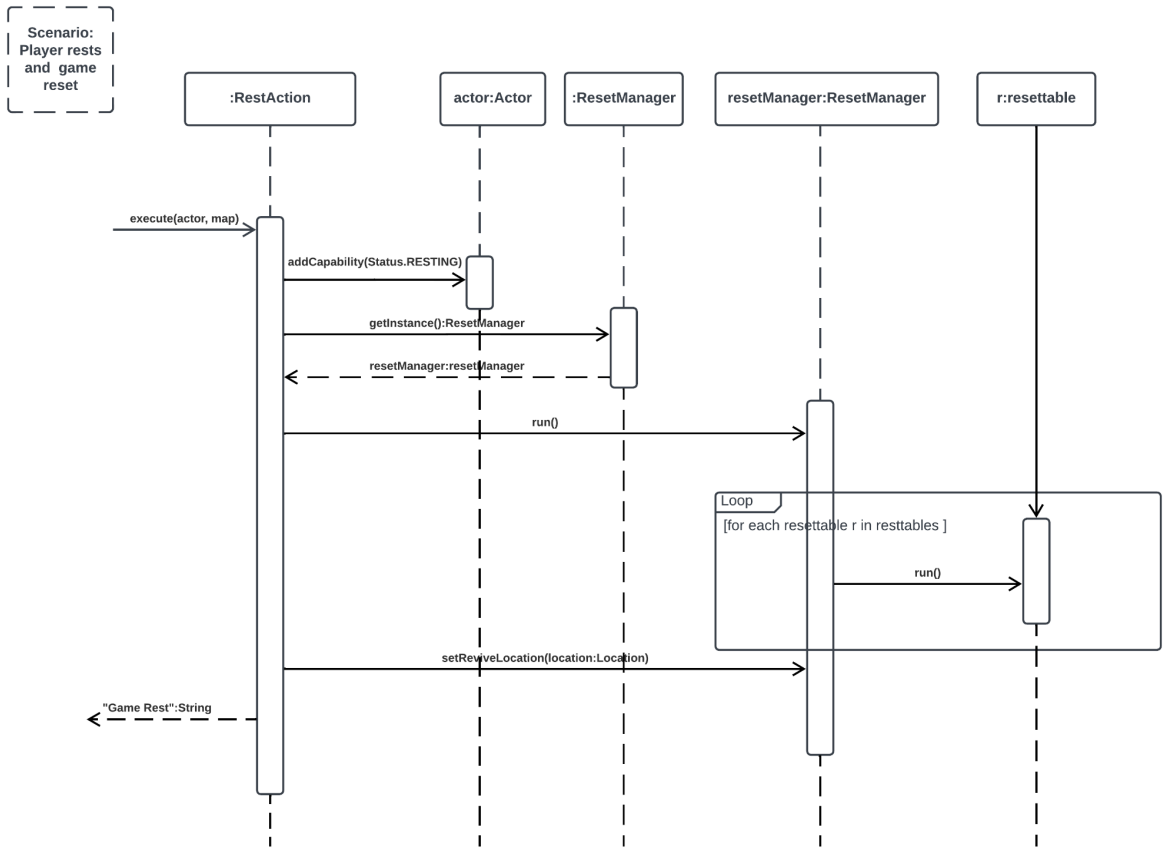
Changes in Assignment 2 for Requirement 3



Changes in Assignment 2:

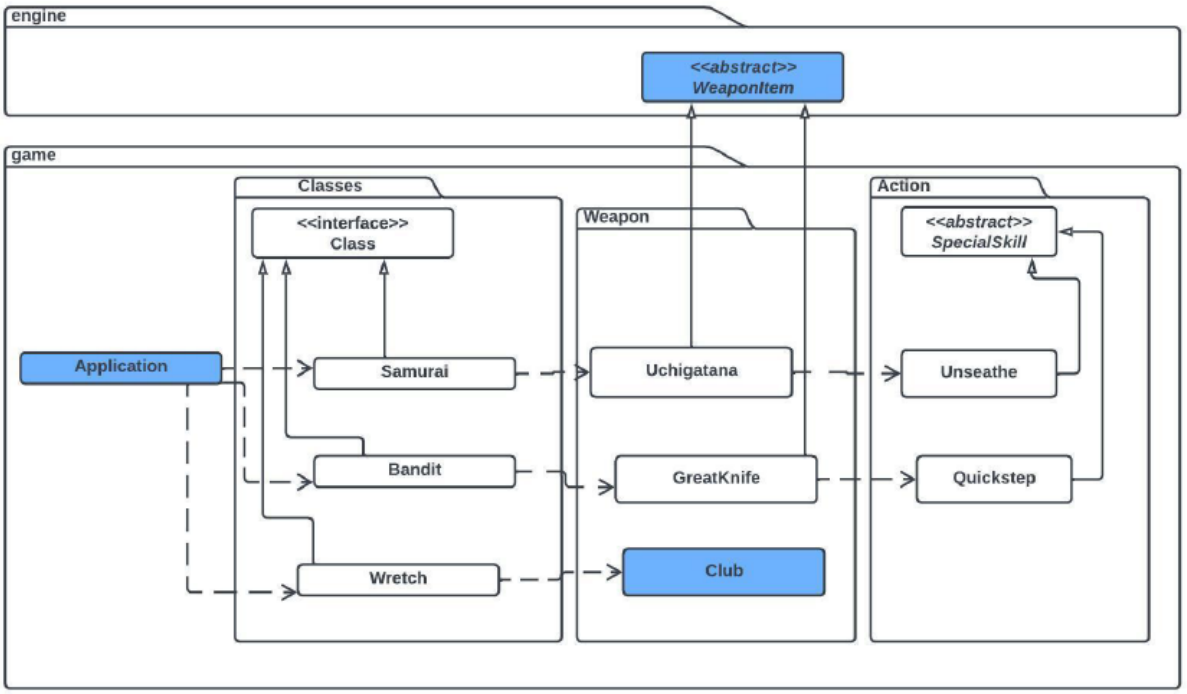
- RecoverRunesAction is updated to extend the action class instead of pickupAction as it can only recover runes items. The benefit of this is it follows the single responsibility principle as it is only responsible for recovering runes. The drawback is that it might be better if we inherit pickupAction 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



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

Requirement 4



Rationale:

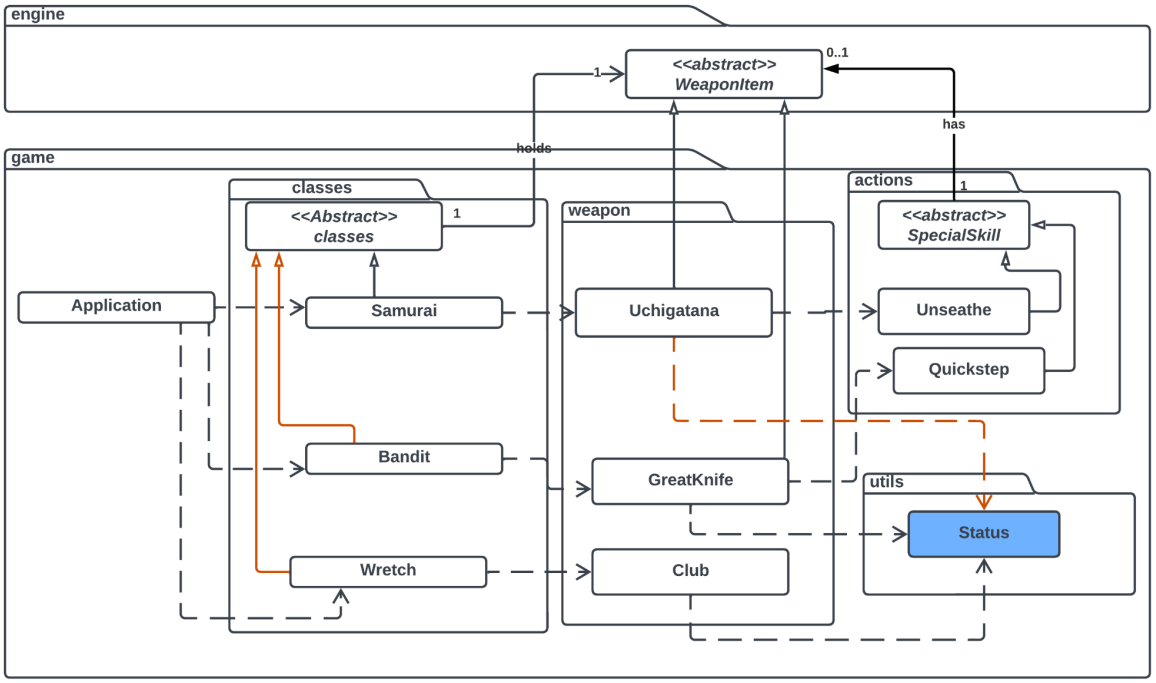
Classes and Combat Archetypes

- 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

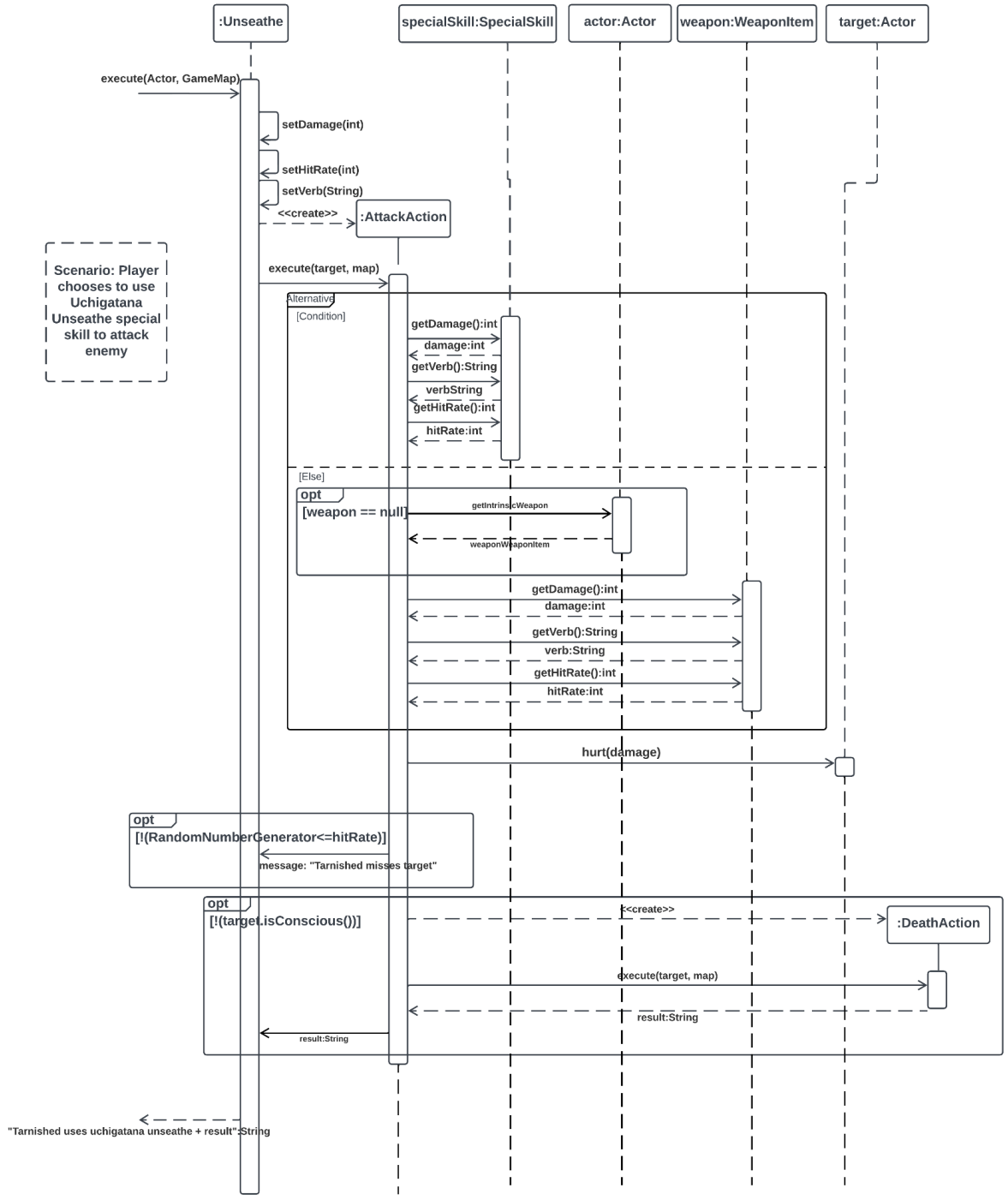
- Uchigatana and Great Knife are both WeaponItem, hence they inherit and extend the WeaponItem class to follow the don't repeat yourself principle.
- Uchigatana has a dependency on Unseathe because it has Unseathe as its special skill
- Great Knife has a dependency on QuickStep because it has QuickStep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



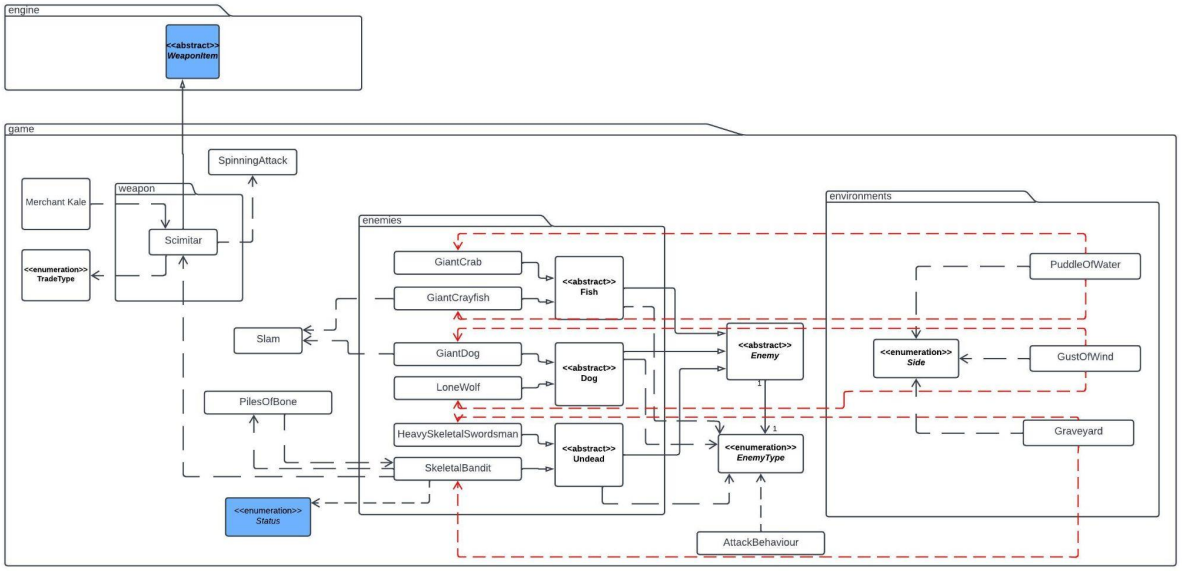
- Dependencies from some weapons to status have been added to indicate whether the weapon has a special skill.
- An association from classes to weaponItem is added because each class has its own starting weapon. The pros it that it will be easy for the application to get the weapon and assign it to the 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.

Requirement 5



Rationale:

More enemies:

Subclasses (abstract classes) of Enemy abstract class

- 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.

Scimitar

- 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).

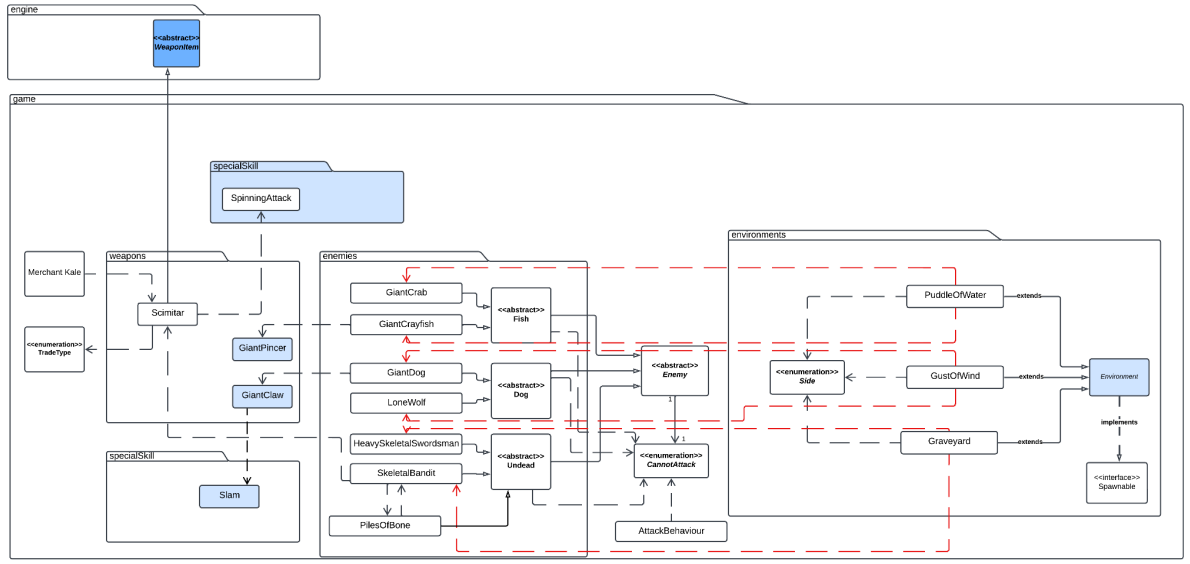
MerchantKale

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

PuddleOfWater, GustOfWind, Graveyard

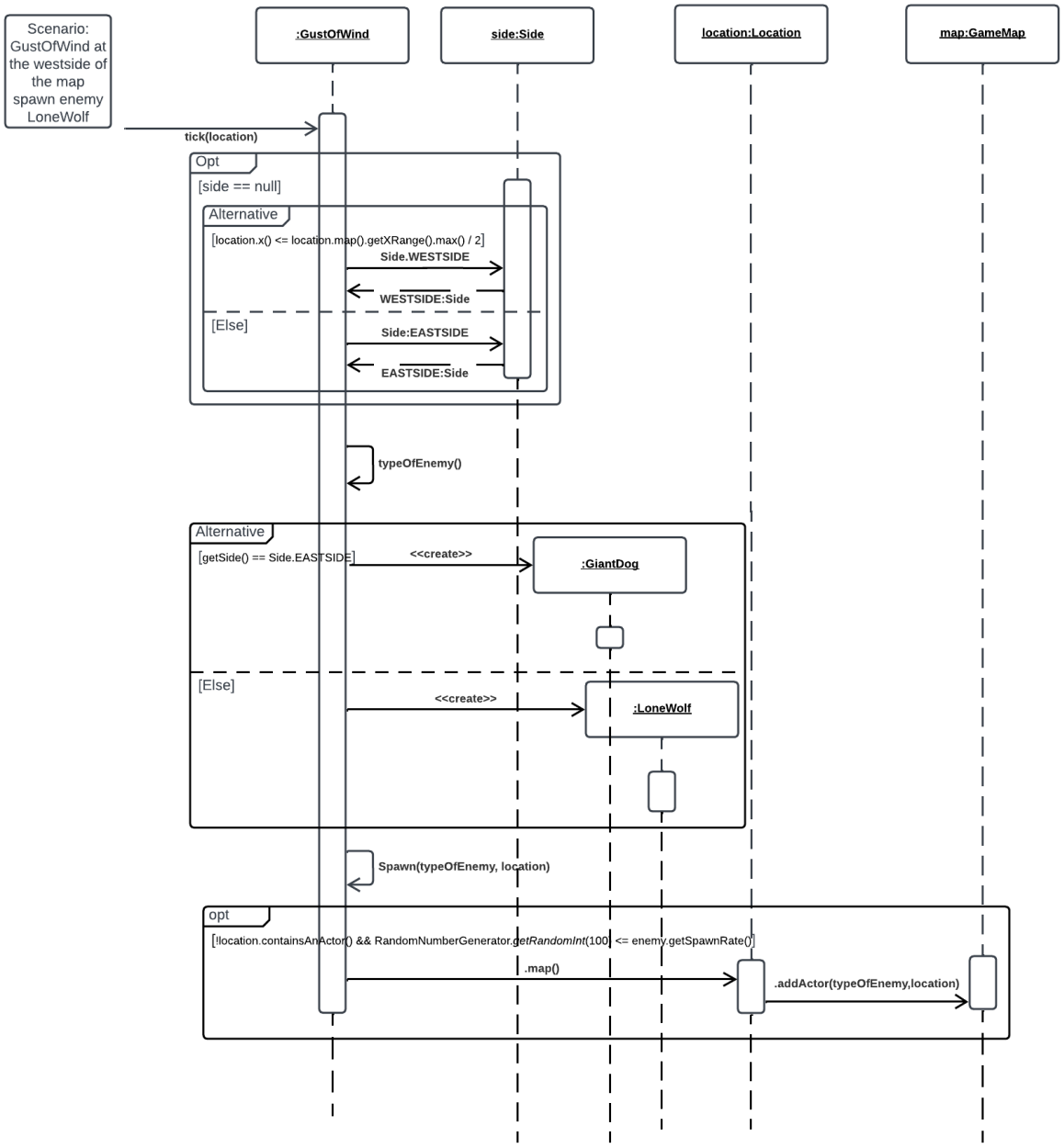
- PuddleOfWater has a dependency on GiantCrab and GiantCrayfish because PuddleOfWater will spawn GiantCrab if it is on the west side of the map, spawn GiantCrayfish if it is on the east side of the map.
- GustOfWind has a dependency on GiantDog and LoneWolf because GustOfWind will spawn LoneWolf if it 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).

Changes in Assignment 2 for Requirement 5



- 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



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