Lab03Group03 A2 Diagram and Design Rationale

Contribution log:

https://docs.google.com/spreadsheets/d/1itMLRX1DOQDyBiHIsicFYc-CcjWs0eaZE00CWoSnUc8/edit?usp=sharing

Requirement 1

Environment

Within the overridden tick() method of an environment, the environment uses a random number generator to generate a number, if this number is then the percentage chance to spawn specified in the spec, the environment will spawn the enemy associated with it.

Enemy

The abstract enemy class has a hashmap that maps a priority as an int to a Behaviour type object, the lower the priority int, the higher the priority the Behaviour has, meaning its getAction will be executed first. Each enemy type will determine its own behaviour and playTurn, the default playTurn would work for common behaviour for all enemies in the specification. Since all enemy has behaviours, this approach allow the most reusability, if we set up some default behaviour in the Enemy abstract class constructor, if we want to make an enemy without those default behaviour (i.e passive enemy who doesn't follow but does attack, pile of bones, etc) we would have to remove them before we set the custom behaviour, which would violate connascence of execute.

To implement the common behaviour, the priority for the Behaviour would be: AttackBehaviour, FollowBehaviour*, DespawnBehaviour, WanderBehaviour

*FollowBehaviour is only added to the behaviour hashmap after the enemy comes close to the player. If we add FollowBehaviour right from the start, the enemy would immediately follow the player once they spawn, no matter where the player is.

Enemy types are stored in the EnemyType enums, enemies with the same capability for EnemyType will not attack each other. There is a static method in Enemy to check for this.

PileOfBone

Enemy of type skeleton in addition to the behaviours above, has a SpawnBehaviour that will remove them from the map and spawns in a PileOfBone.

PileOfBones only has a ReviveBehaviour which will revive the original skeleton after a set number of turns. If it is hit, it will drop the inventory of the original skeleton.

AttackBehaviour

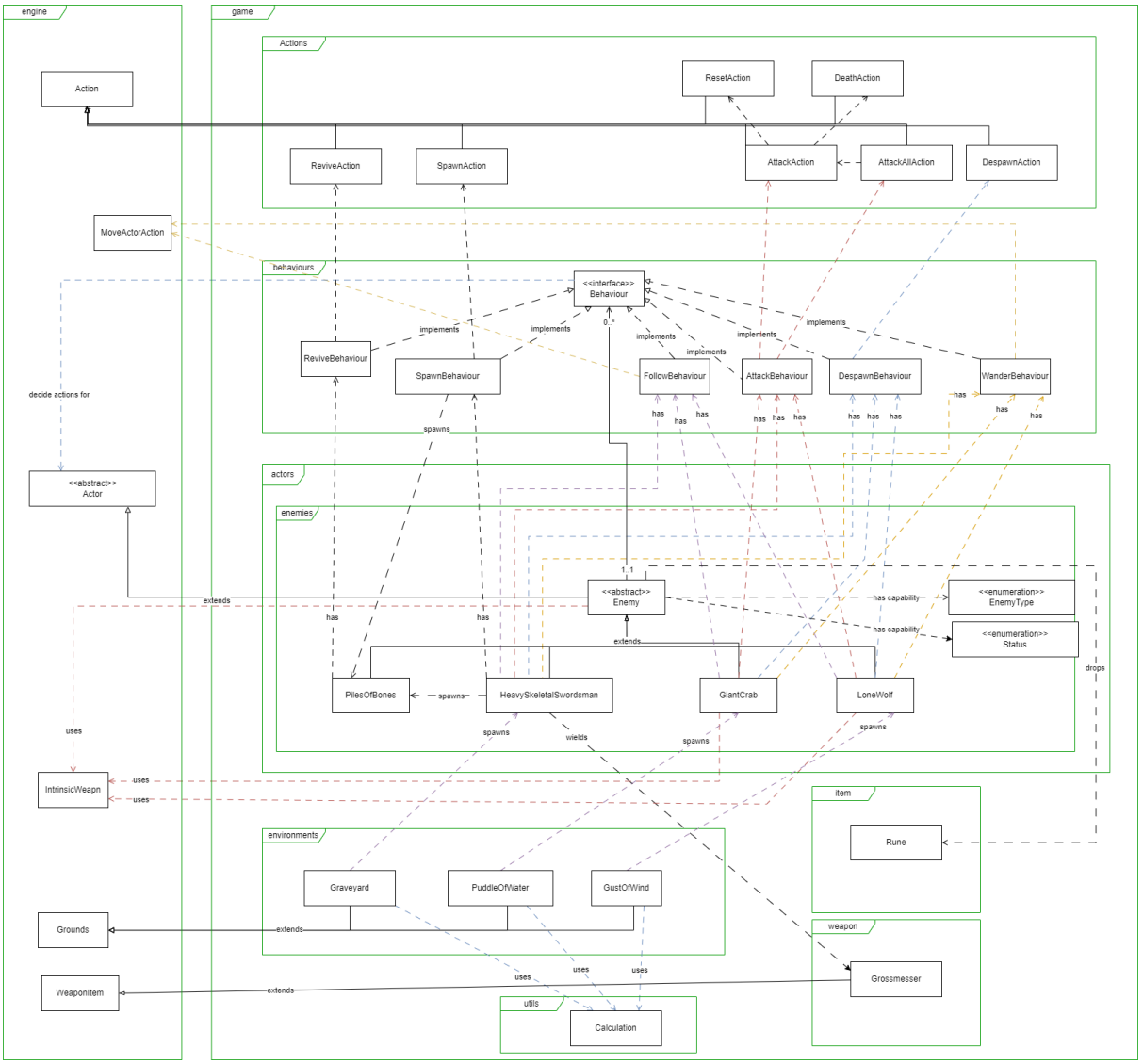
AttackBehaviour constructor takes 1 argument: canAttackAll, if an enemy has this behaviour with this argument set to true, they have 50% chance to attack the surrounding with their IntrinsicWeapon. We use this approach instead of implementing a getSkill for a weapon because intuitively, these enemies aren't attacking with weapons, and since multiple

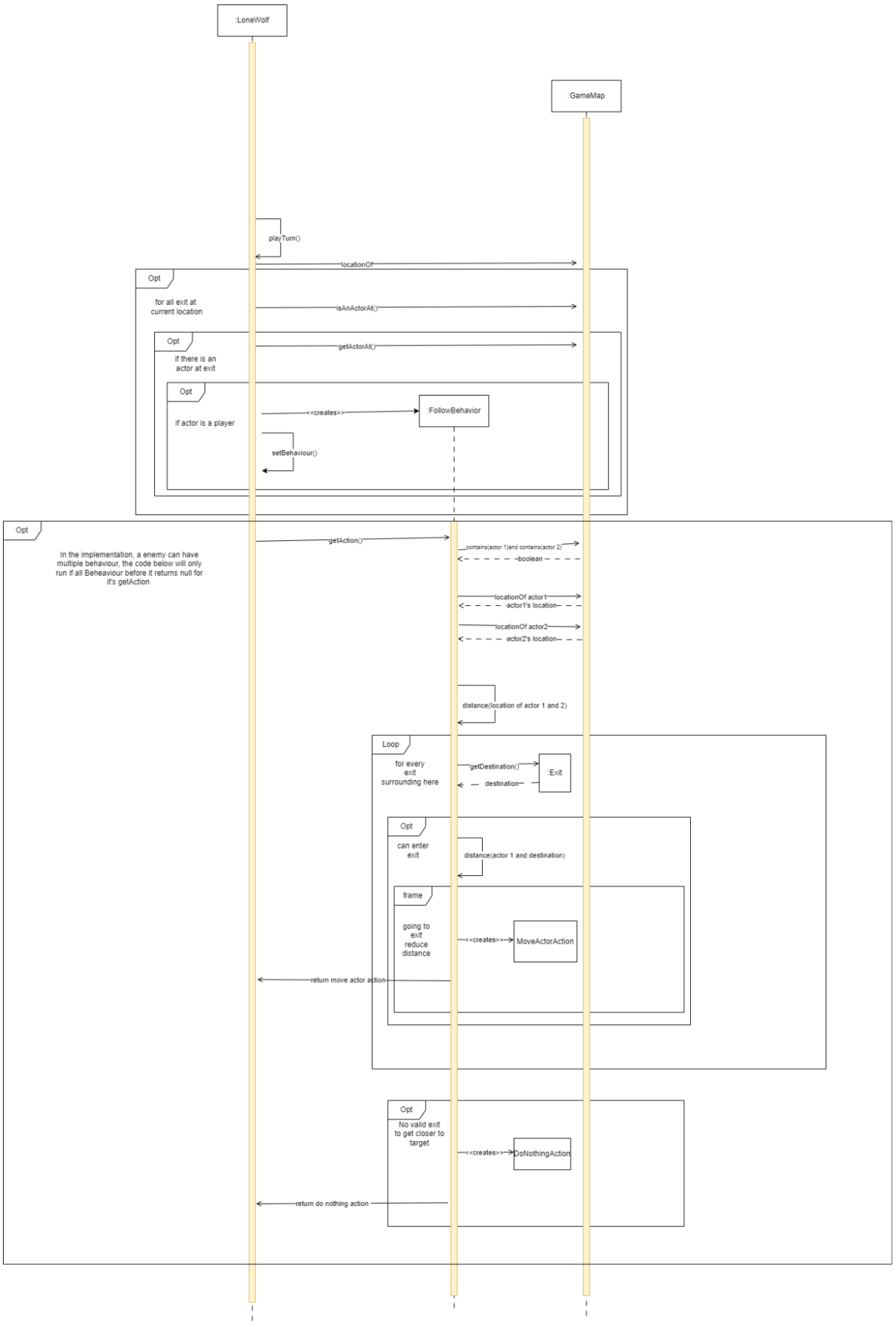
enemies can “slam”, we would have to implement multiple weapons that are almost identical and can only be used with 1 enemy, adding bulk and reducing reusability.

“Slams” are represented by the AttackAllAction, which creates and executes multiple AttackAction. If AttackAction is executed onto an enemy, a DeathAction is created and executed, otherwise ResetAction for the player.

DeathAction takes care of events that happens at an actors death, i.e dropping item, removing actor, etc

DespawnBehaviour: has a 10% chance of executing a DespawnAction, removing the actor from the map
FollowBehaviour and WanderBehaviour algorithms are left as it is.




:LoneWolf
:GameMap
playTurn()
locationOf
Opt
for all exit at current location
isAnActorAt()
Opt
getActorAt()
if there is an actor at exit
Opt
if actor is a player
<<creates>>
:FollowBehavior
setBehaviour()
Opt
In the implementation, a enemy can have multiple behaviour, the code below will only run if all Beheaviour before it returns null for it's getAction
getAction()
contains(actor 1)and contains(actor 2)
boolean
locationOf actor1
actor1's location
locationOf actor2
actor2's location
distance(location of actor 1 and 2)
Loop
for every exit surrounding here
getDestination()
destination
:Exit
Opt
can enter exit
distance(actor 1 and destination)
frame
going to exit reduce distance
<<creates>>
MoveActorAction
return move actor action
Opt
No valid exit to get closer to target
<<creates>>
DoNothingAction
return do nothing action

Requirement 2

BuyerSellerList is a class that lists actors that are allowed to purchase and sell weapons to traders. This class makes use of factory methods to only have one instance by associating with itself. This class has an association with the BuySellCapable interface to ensure that only actors implementing the interface can be stored. This class helps Trader determine if the actor next to it could purchase and sell weapons or not. This adheres to Liskov Substitution Principle (LSP) because methods meant for selling and buying will only be invoked on actors that implement BuySellCapable and not invoked on other actors and get unexpected output. The benefit is that if the game is multiplayer more players implementing BuySellCapable can be added to BuyerSellerList to allow them to buy and sell weapons.

Trader has a dependency on SellAction (sell weapon) and BuyAction (buy weapon) actions. This allows the Trader to allow these actions to be performed on it by the Player. SellAction and BuyAction will be shown as sell weapons and buy weapon options in the menu for the Player when the Player is near the Trader. The Trader depends on Status and BuyerSellerList to check if the actor is a Player. The SellAction and BuyAction inherits from Action abstract class to reduce code repetition adhering to DRY. Trader has an association with sellable and purchaseable interface to be able to store weapons that can be sold and purchased by player. This adheres to Liskov Substitution Principle (LSP) because methods meant for weapons that can be sold or bought are only invoked on weapons implementing Sellable or Purchasable ensuring that only weapons that can be bought and sold are stored and have methods that run as expected. Sellable and Purchasable interfaces depend on WeaponItem to be able to return the weapon of purchase or sale. This makes the game extensible if more weapons can be sold or purchased they just need to implement Sellable and Purchasable interfaces to be added to Trader. Another benefit is that if more traders are added to the game they can make use of purchaseable and sellable interfaces to be able to allow players to sell and buy specific weapons from them. The con is that there will be many copies of the lists of these weapons, we could use a single manager if all traders allow selling and purchasing of the same weapons but this will not be suitable if each trader has its unique weapons that are allowed to sell and buy.

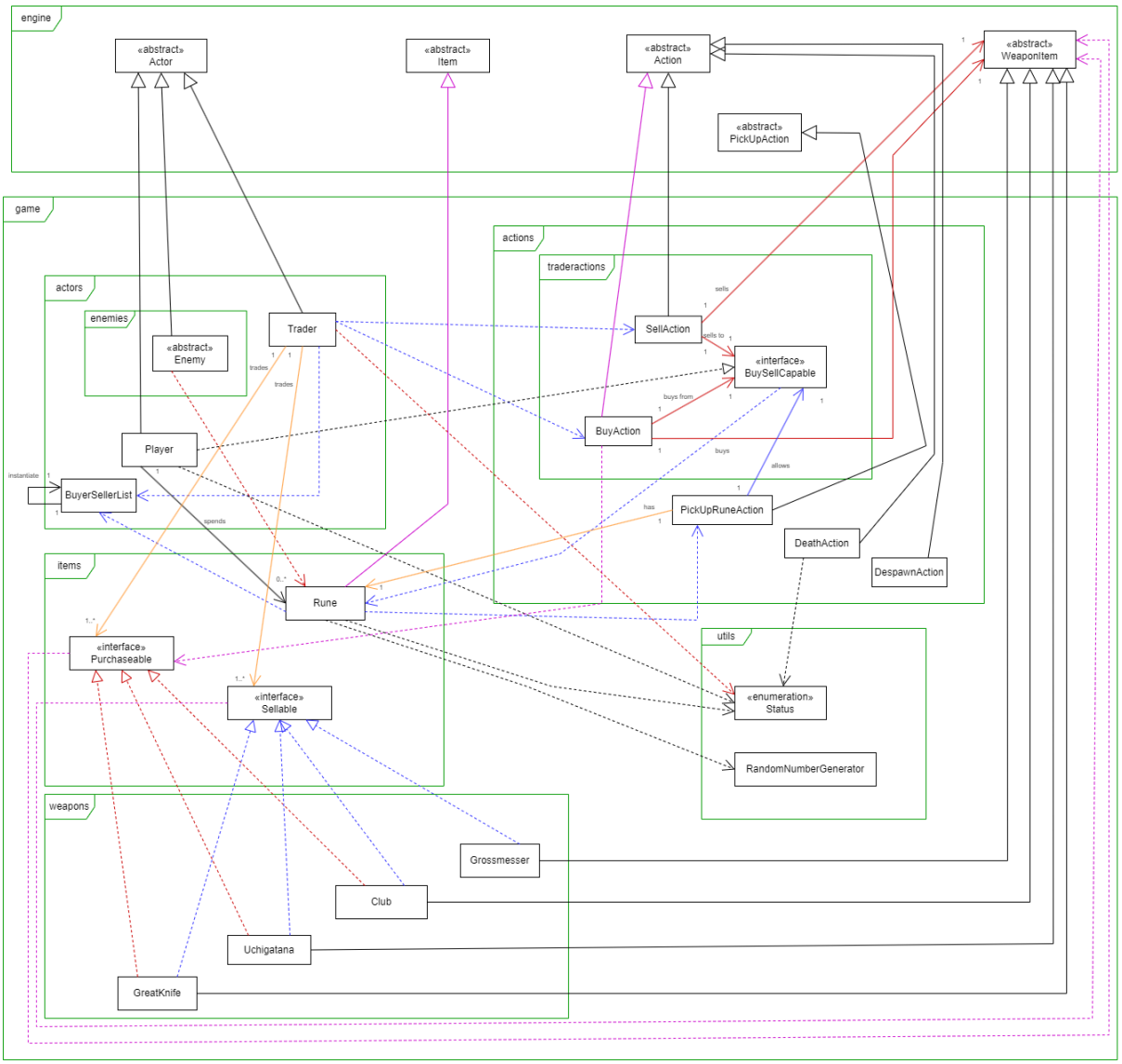
Rune inherits from item abstract class in the game engine. They need to be able to be dropped and picked up by the player and enemies should drop them. Item class provides these characteristics and by inheriting it code repetition is reduced adhering DRY. The Rune class depends on PickUpRuneAction to allow it to be picked up by the Player. The benefit is that code repetition is reduced by inheriting from Item abstract class. However, the con is that Rune has a lot of unique features and storing it as an Item in the game makes it difficult to make use of the unique features of Rune class. The Rune class depends on Status to help DeathAction check whether it is a Rune or not to determine whether to drop the Rune from the killed enemy to the ground or pass it to the attacker who is a Player. The Rune class depends on RandomNumberGenerator to help it randomly assign Rune values to enemies.

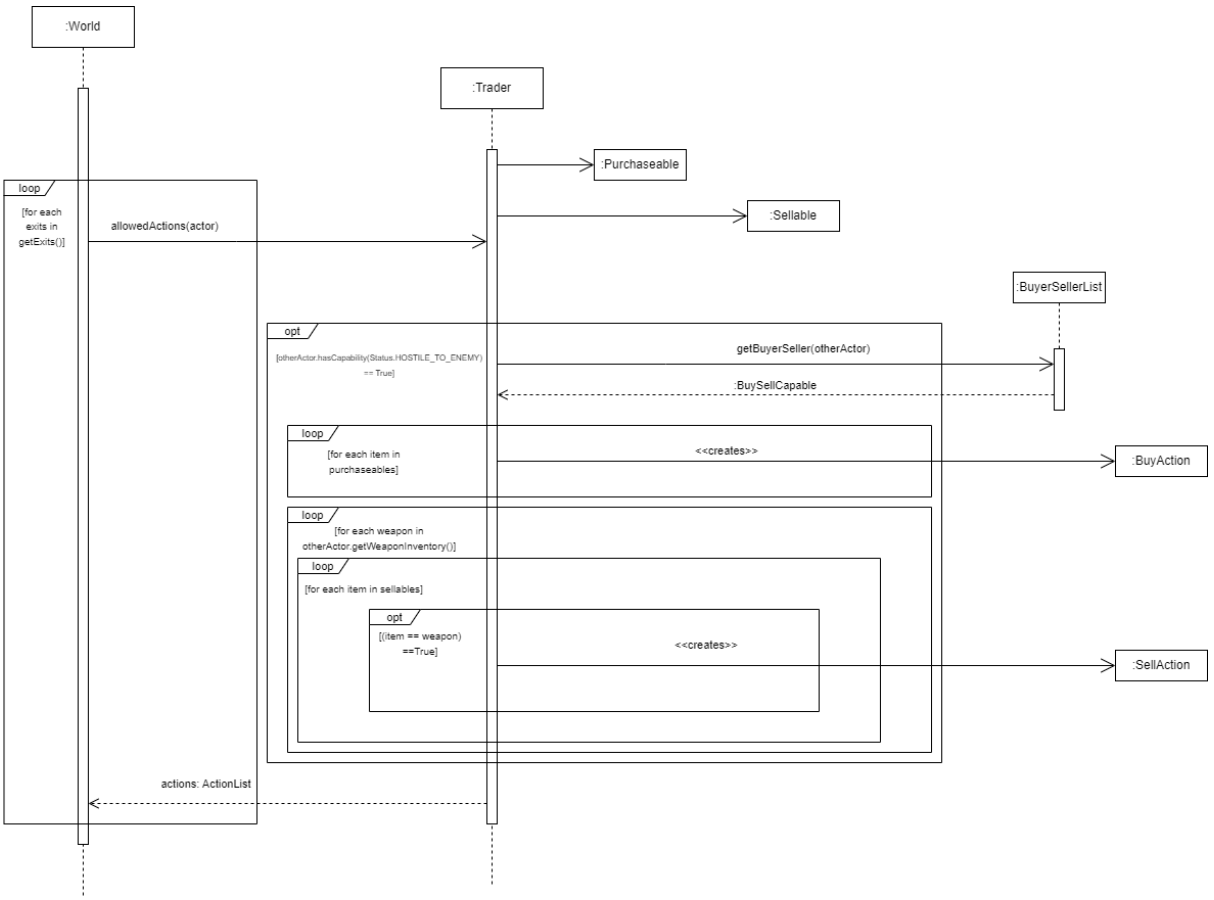
The DeathActions which are executed if enemies are killed have a dependency on Status to help it check if the Item to be dropped when enemy is dead is Rune or not. If it is a Rune,

DeathAction will pass in the attacker to the Rune getDropAction so that it could be checked if the attacker is a Player that implements BuySellCapable using BuyerSellerList to directly give the Rune to player using else it would drop the Rune to the ground like other items. This implementation is beneficial because the Rune class takes responsibility to check if the attacker is the Player or not and if any logic needs change the Rune class can be modified adhering to Single Responsibility Principle (SRP) where Rune has code for its responsibilities. The bad thing is that if there are multiplayers and some players have different logic for Rune dropping we need a separate list of Players to be able to determine how the Rune is dropped. The DespawnAction directly removes the enemy from the map if the enemy is despawned so its Rune does not get dropped or given to the Player.

The Player implements the BuySellCapable interface. The Player needs extra code to handle buy and sell actions of weapons with Trader using Runes without affecting its existing functionality. Hence, by implementing the BuySellCapable interface the code in Player does not need to be modified and it just gets added functionality which adheres to the Open-Closed Principle because the existing classes are not modified and just extended with features using interfaces. The Player has an association with Rune so that it can store the Rune separately and be able to use its unique methods to assist it with making purchases and selling weapons to a trader. Also this helps the Player to directly receive Rune when it kills enemies. This is beneficial because any logic for Rune can be controlled separately and any changes are easy to modify.

Enemy abstract class depends on Rune class because it needs to add Rune to its inventory so that they can be dropped when they die. The Rune was inherited from the item so that it can make use of existing drop item related methods in actor class to drop runes when enemies die. This is beneficial as no extra code is needed reducing code repetition.


engine
«abstract» Actor
«abstract» Item
«abstract» Action
«abstract» WeaponItem
«abstract» PickUpAction
game
actors
enemies
«abstract» Enemy
Trader
Player
BuyerSellerList
instantiate
trades
spends
actions
traderactions
SellAction
BuyAction
«interface» BuySellCapable
sells
sells to
buys from
buys
allows
PickUpRuneAction
has
DeathAction
DespawnAction
items
Rune
«interface» Purchaseable
«interface» Sellable
utils
«enumeration» Status
RandomNumberGenerator
weapons
Grossmesser
Club
Uchigatana
GreatKnife



Requirement 3

Resettable interface is implemented by actors, items etc that should be reset when the game is reset. Resettable interface has dependency on Actor and GameMap class to be able to determine the actor and location performing the reset action. These resetables are stored in the ResetManager having association with Resettable that contains code to perform reset actions on all the resetables. Only classes that are a subtype of the Resettable interface can be added to Resetmanager class to ensure that only the reset methods of objects that contain them are called. By doing so we can be sure that the objects in ResetManager have functioning reset methods that can be called without causing errors in the program when the game needs to be reset and this adheres to the Liskov Substitution Principle (LSP). This approach is good because it can prevent errors if reset methods are invoked on objects that do not need any change when the game is reset. However, it takes up extra memory because a list of these resettables must be maintained throughout the game.

ResetAction in the reset package is inherited from the action abstract class. AttackAction has a dependency with ResetAction and Status so that if the target being killed is the Player (checked using Status) this action will be returned by the Player if the player gets killed to trigger the reset functionality of the game. ResetAction has a dependency on ResetManager. If a new actor, item and etc are resettable added to the game, code does not have to be modified in ResetAction as these resetables will be managed by ResetManager adhering to the Open-Close Principle. ResetAction depends on FancyMessage to be able to print a message if the Player dies. The positive thing is that if any reset requirements change for

any game object, we just need to modify the methods they get from implementing Resettable interface, this makes it easier to modify as only one place needs to be modified. More game objects that need to change in game reset can just implement Resettable interface and be added to ResetManager without the need to modify ResetAction and the game will work as expected because they have Resettable interface. The negative thing of this approach is that at certain resets only certain game objects out of all resettable should be reset wouldn't work because the current way would reset any object that is resettable. The RestAction has dependency on ResetAction so that if Player chooses to rest at Site of Lost Grace the game can be reset without any extra code. This reduces code repetition adhering to DRY.

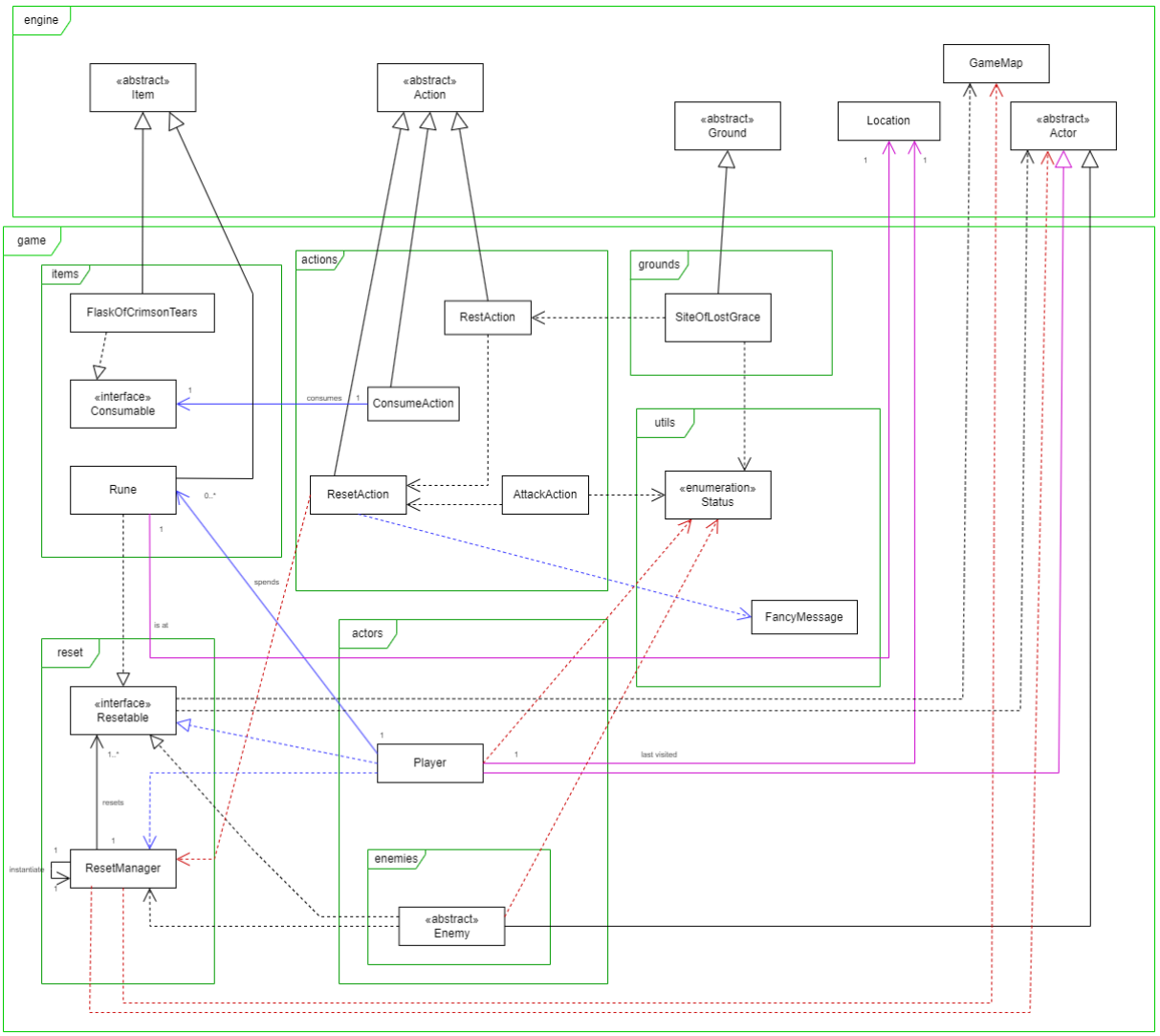
FlaskOfCrimsonTears inherit from item abstract class in the game engine. FlaskOfCrimsonTears share the same characteristics of an item. The FlaskOfCrimsonTears implements the Consumable interface to have extra code to help the Player consume it at a max of 2 times. Moreover, the ConsumeAction has an association with the Consumable interface to allow the Player to only be able to choose to consume items that can be consumed such as FlaskOfCrimsonTears. This helps adhere to LSP because if only if the item implements Consumable interface they can be passed into ConsumeAction and relevant methods for consuming items from Consumable interface can be invoked with expected output with no error. This makes the game more extensible because if more items are consumable in the future they just need to implement Consumable interface to be able to be used by ConsumeAction, they can have their own rules on consumption but work as expected without needing to modify code at multiple places.

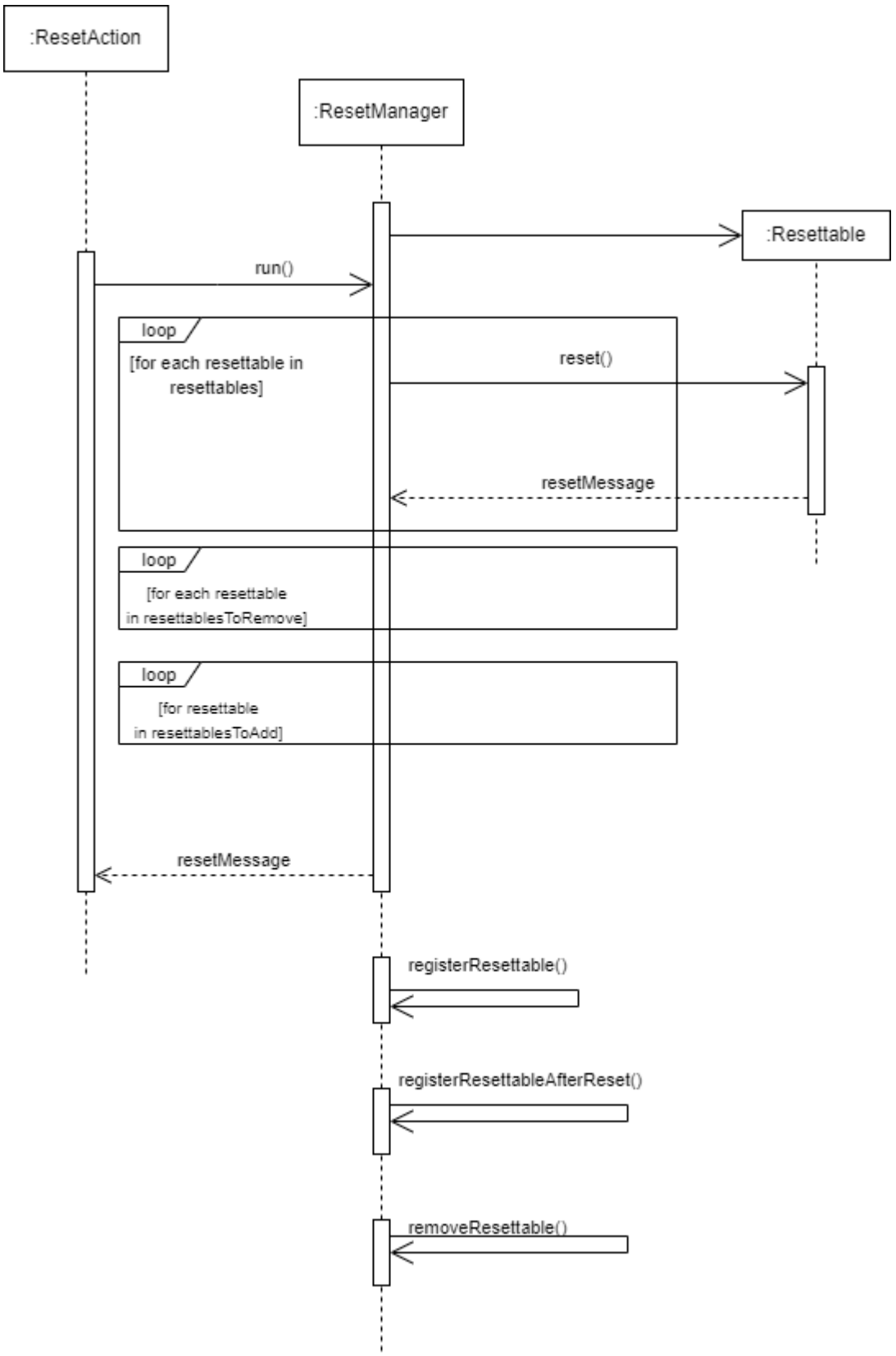
SiteOfLostGrace inherits from the ground class in the game engine. SiteOfLostGrace shares the same characteristics of a ground. By inheriting from abstract classes code repetition is reduced adhering to the Do Not Repeat Yourself (DRY) principle. The SiteOfLostGrace has a dependency on RestAction and Status to be able to determine if the nearby actor is a Player and allow it to rest on it. By doing so only the Player can rest on it and not other enemies.

Rune inherits from item abstract class in the game engine. They need to be able to be dropped and picked up by the player and enemies should drop them. Item class provides these characteristics and by inheriting it code repetition is reduced adhering DRY. Rune class implements resettable so that Rune objects can be removed if the player dies again (ResetAction triggered) after they were dropped onto a ground. The Rune class has an association with Location so that when it is dropped to ground it can save its Location to be used during game reset to remove itself from the ground. This helps keep all the reset responsibility of a Rune in the Rune class adhering to Single Responsibility Principle (SRP). This makes code more extensible and easier to maintain because each class has a single responsibility. This makes it hard to read because there will be many classes with single responsibility.

When enemies inherit the enemy abstract class they can store capabilities in the form of enums. Status enum is used to identify enemies that are spawnable. By doing so the use of instanceof and magic strings that can break the program can be avoided. Enums help ensure consistency when checking for a certain type. Enemy and Player depend on ResetManager and implement Resettable interface. This helps them have their own implementations of reset when the game is reset. This helps standardise the methods used

for reset adhering to LSP. Makes the game more extensible as every new enemy will be able to have its own logic for reset. Player has association with Location to be able to record the last visited SiteOfLostGrace. This is beneficial if in future there are many SiteOfLostGrace, the Player code doesn't need modification as it can deal with many SiteOfLostGrace using Status dependency. The Player class has association with Rune to be able to have specific code to determine whether the Rune should be dropped or not if it dies or rest. This helps maintain the responsibility of Player Rune dropping inside the Player class adhering to SRP.


engine
«abstract»
Item
«abstract»
Action
«abstract»
Ground
Location
GameMap
«abstract»
Actor
game
items
FlaskOfCrimsonTears
«interface»
Consumable
Rune
actions
RestAction
consumes
ConsumeAction
ResetAction
AttackAction
grounds
SiteOfLostGrace
utils
«enumeration»
Status
FancyMessage
spends
is at
actors
reset
«interface»
Resetable
Player
last visited
resets
instantiate
ResetManager
enemies
«abstract»
Enemy


:ResetAction
:ResetManager
:Resettable
run()
loop
[for each resettable in resettables]
reset()
resetMessage
loop
[for each resettable in resettablesToRemove]
loop
[for resettable in resettablesToAdd]
resetMessage
registerResettable()
registerResettableAfterReset()
removeResettable()

Requirement 4

The requirement 4 UML diagram represents the implementation of the combat archetypes and the associated methods and skills that are associated with those archetypes. The archetypes determine the starting hitpoint of an actor, in this case the player, as well as the starting weapon and the skill associated. This diagram reflects the implementation of the code in the game, and the following rationale will discuss the implementation and changes made to implement the requirement.

Firstly, following the comments of the previous design feedback, we implemented an abstract class called Archetype, from which we can extend different combat archetypes classes from and initialise their respective hit points and weapon. In this case, we have the Bandit, Wretch and Samurai class all extending from Archetype. While the previous design had ArchetypeManager storing the different combat archetypes attributes, this design separated the combat archetypes. The benefits from this is that the archetype is much more open to extension, particularly if we want archetypes to be accessible for other actors. If we used the old design, this extension would violate the Open-Closed Principle, because we would need to directly modify ArchetypeManager to access the archetypes. Additionally, the sub-classes extending from Archetype all have an association with a weapon that corresponds to that archetype, and they also hold the hit point which the actor would have, if that combat archetype was selected. For example, Bandit uses a GreatKnife.

Moving on, the ArchetypeManager still has the task of setting up the Player through the createPlayer() method. However, instead of storing the combat archetype details, the ArchetypeManager has an association with each combat archetype, creating an instance of these archetypes and storing them in a dictionary for use. The ArchetypeManager class also has another method called selectArchetype() which requires UI to return the selected archetype in the form of a key. This key is used by createPlayer() to access the instance of the combat archetype that can be used to make the player.

From there, the Application class can call on the ArchetypeManager to run and set up the Player instead of initialising the Player in the Application class. This follows the Single Responsibility Principle (SRP) whereby the Application class should focus on running the main code for the world, instead of also having to determine the Player's archetype.

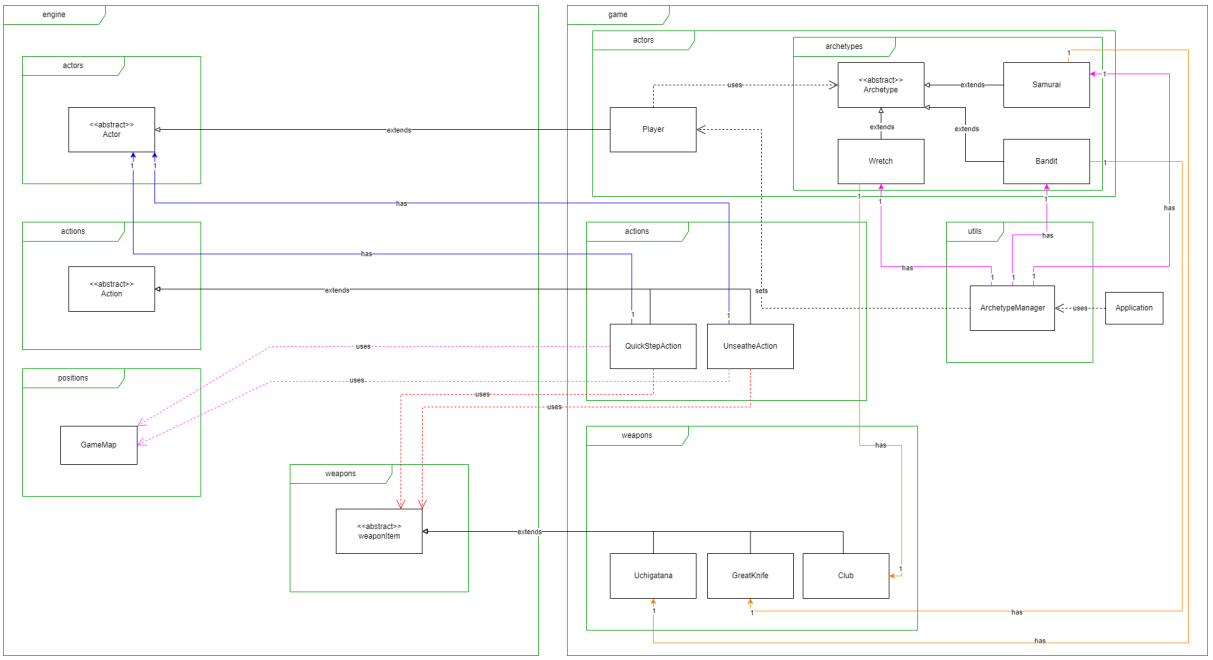
Moreover, the Player constructor is modified to include the archetype, where the player's hitpoint and weapon can be accessible from the input archetype. Hence, the Player has a dependency on Archetype class.

The Player will be directly extended from the abstract class Actor (from the engine) as it has the similar features and attributes. This will reduce repetition of re-coding the attributes and methods (DRY).

The special skills, i.e. QuickStepAction and Unseathe Action, which are associated with the weapons of the unique combat archetype, will be extended from Action (DRY). These skills will be accessible to the player via the getSkill() implementation within the weapon. These

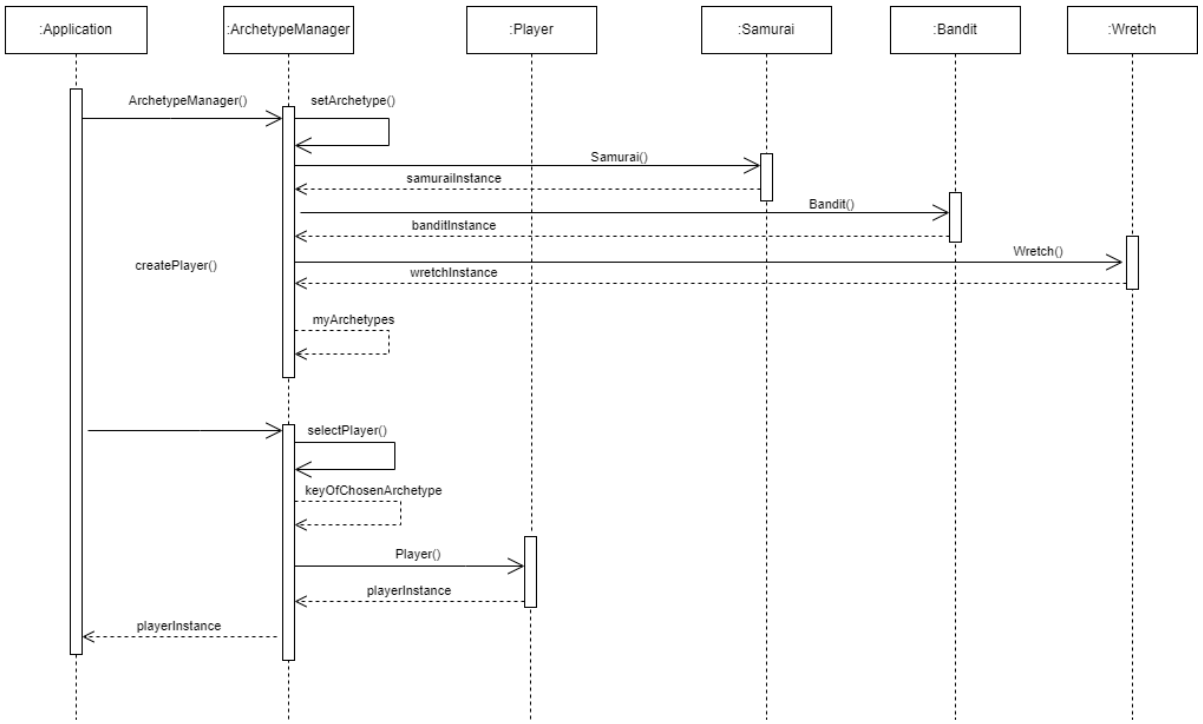
actions also rely upon the GameMap, to identify the target, movement, attack, etc. Also each action has an associated Actor that represents the target.

The unique weapon associated with each archetype, i.e. Uchigatana, GreatKnife and Club, will be extended from the abstract Weapon class (DRY), and the special skill associated with these weapons will be implemented.



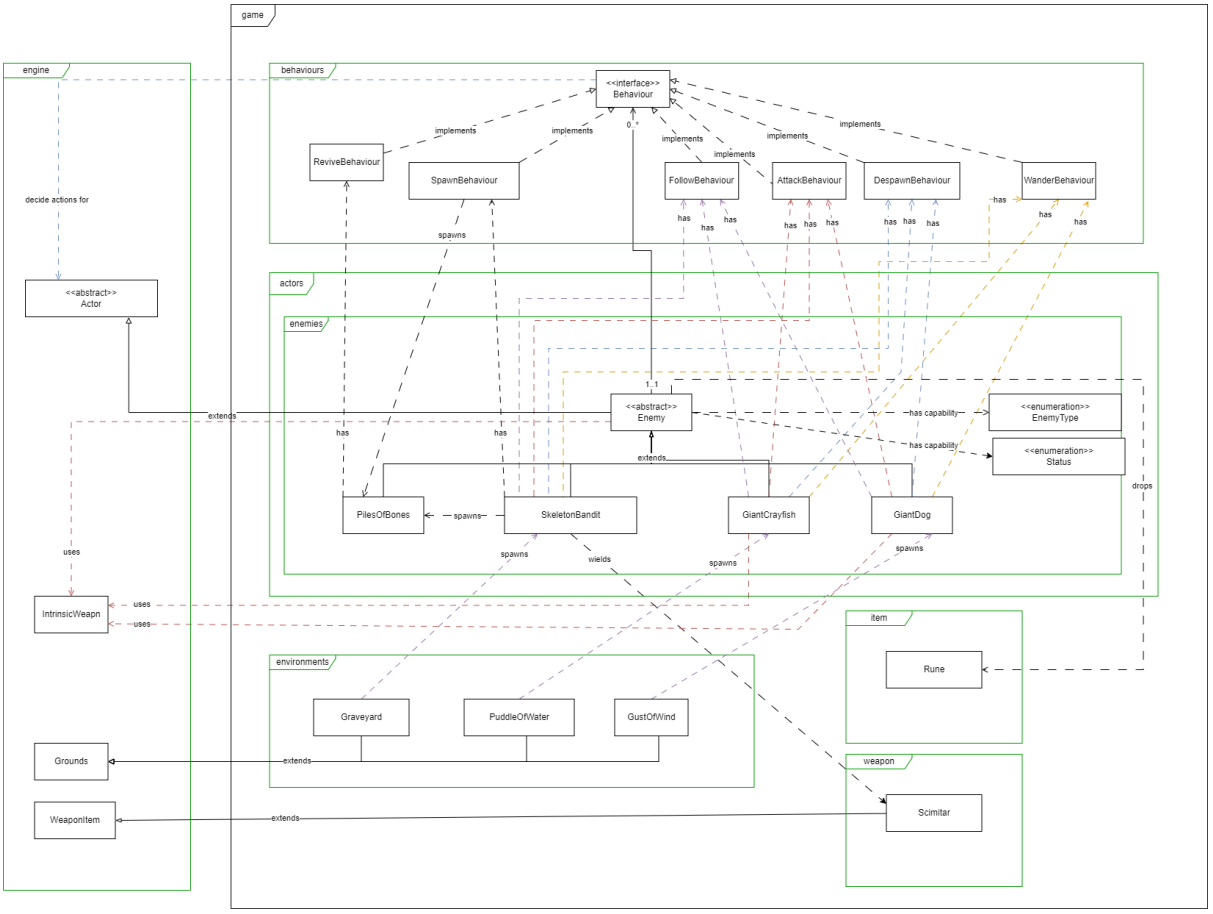
Sequence Diagram for How Archetype is Set

When creating archetype manager, instance of different archetypes are created and added into a dictionary. When the user input, the input is checked against the key of the archetypes in the library. Base on the key chosen, the archetype associated with that key will be returned.



Requirement 5:

In Calculation in the util package, there is an abstract method which determines whether a location is on the east or west side of the map. It does this by checking whether the x coordinate is greater or lesser than half of the map length.
The environment uses this in an if-else clause in the overridden tick() method to implement the spawning of their respective enemy for the east and west side.
SkeletonBandit's implementation is highly similar to HeavySkeletionSwordman except for the weapon and some stats that is given to it on spawn
GiantDog's implementation is highly similar to LoneWolf except for that it can slam (canAttackAll of its AttackBehaviour is set to true) and some stats that are given to it on spawn.
GiantCrayfish's implementation is highly similar to GiantCrab except for some stats that are given to it on spawn.
The fact that all additional enemy types in Requirement 5 doesn't require a lot of extra extension from Requirement 1 proves the high reusability of our enemy design approach.



Sequence Diagram for how the AttackBehaviour works in GiantDog

Works similarly in other enemies, explains how this behaviour determines when is it valid to return an attack and which attack to return