REQ 1 Design Rationale

The diagram represents the first requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring.
We have three design goals for this requirement: implementing environment, implementing enemies, and implementing weapons.

For the environment; Graveyard, GustOfWind, and PuddleOfWater are all extended from the Ground abstract class. This is due to all three of the environment classes having the same common attributes and methods, so we abstract these classes to avoid code repetition (DRY).

We created a new abstract class called Enemies so that all the enemy classes (HeavySkeletalSwordsman, LoneWolf, GiantCrab) can extend from it and do not have code repetition since they would all have common attributes and methods (DRY). All enemies will have the same set of behaviours. And by using the Behaviour Interface, we can have the Enemy class act on the Behaviour interface rather than depending on each behaviour classes, this way we can easily modify it for future use (DIP).

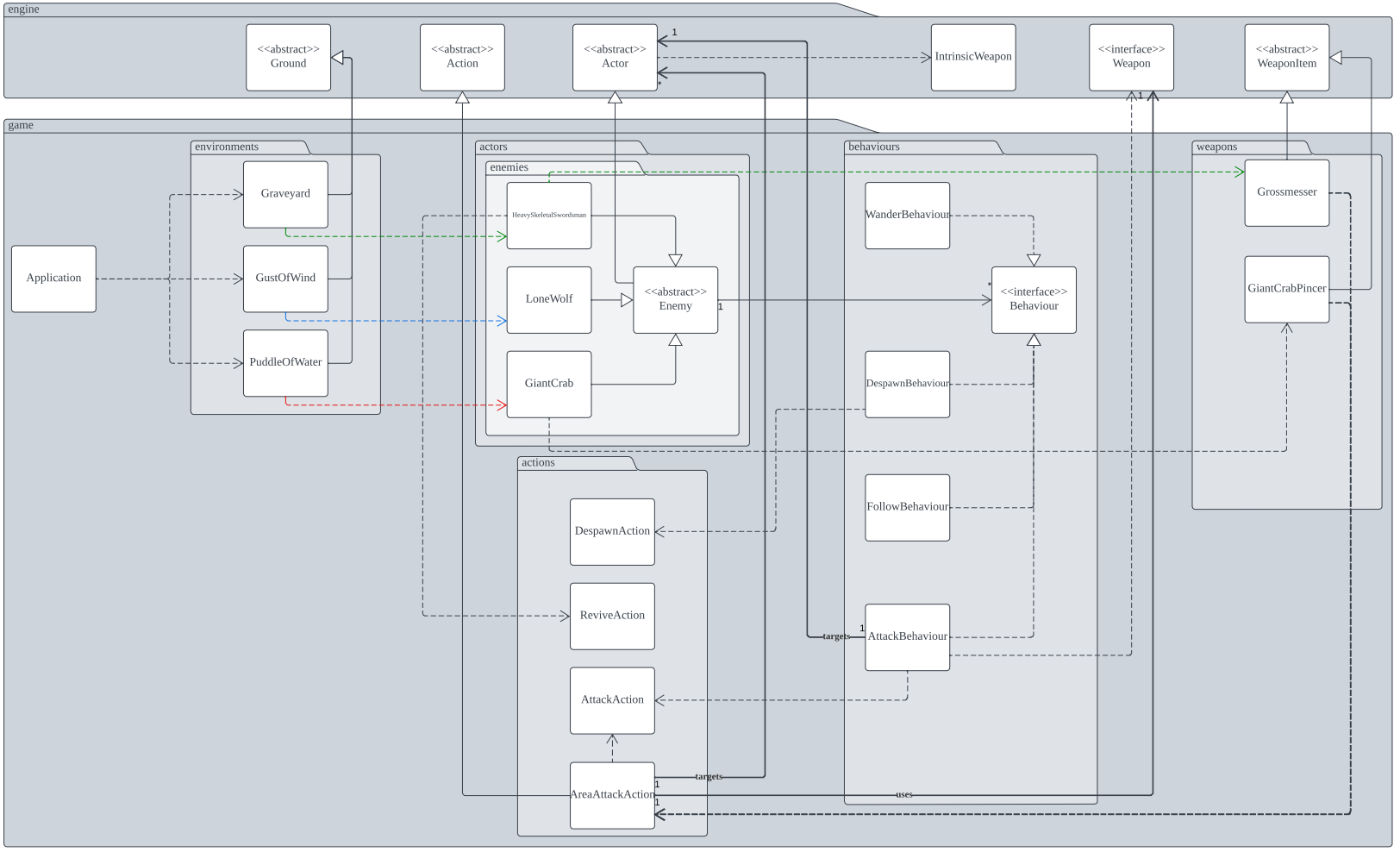
All the behaviour classes are implemented from the Behaviour interface, this is due to class like AttackBehaviour implementing both the Behaviour interface and also the Weapon interface, if both of them were to be classes this wouldn't be able to work. And new set of operations could be easily added to the system by adding more interfaces since each of the current interfaces is separated by its own purpose (Interface Segregation Principle).

Some enemies can decide to either attack a single target or multiple at once, so in order to not let AttackAction having both responsibility, a new class is created to handle attacks on multiple enemies on an area, and let AttackAction to strictly handle single target attacks. (SRP)

Grossmesser extended the WeaponItem abstract class. Since all weapons share some common attributes and methods, it is logical to extend Grossmesser from WeaponItem abstract class so that Grossmesser have all the attributes and methods that a weapon should have without code repetition (DRY).

Changes
Based on the feedback result from the first assignment we have decided to made a couple of changes. In this requirement we removed the PileOfBones class and the ReviveBehaviour class, this is because we decided to use the Status enum and changing the display character into the PileOfBone character ('X') by modifying the HeavySkeletalSwordsman class and adding a new attribute called stateCounter, which keeps track of the number of turns the skeleton is in its PileOfBones form.

We then also package all the classes and interfaces into their respective package for organization, modularity, and encapsulation.


engine
<<abstract>>
Ground
<<abstract>>
Action
<<abstract>>
Actor
IntrinsicWeapon
<<interface>>
Weapon
<<abstract>>
WeaponItem
game
environments
Graveyard
GustOfWind
PuddleOfWater
Application
actors
enemies
HeavySkeletalSwordsman
LoneWolf
GiantCrab
<<abstract>>
Enemy
behaviours
WanderBehaviour
<<interface>>
Behaviour
DespawnBehaviour
FollowBehaviour
AttackBehaviour
weapons
Grossmesser
GiantCrabPincer
actions
DespawnAction
ReviveAction
AttackAction
AreaAttackAction
targets
uses
1
*

REQ 2 Design Rationale

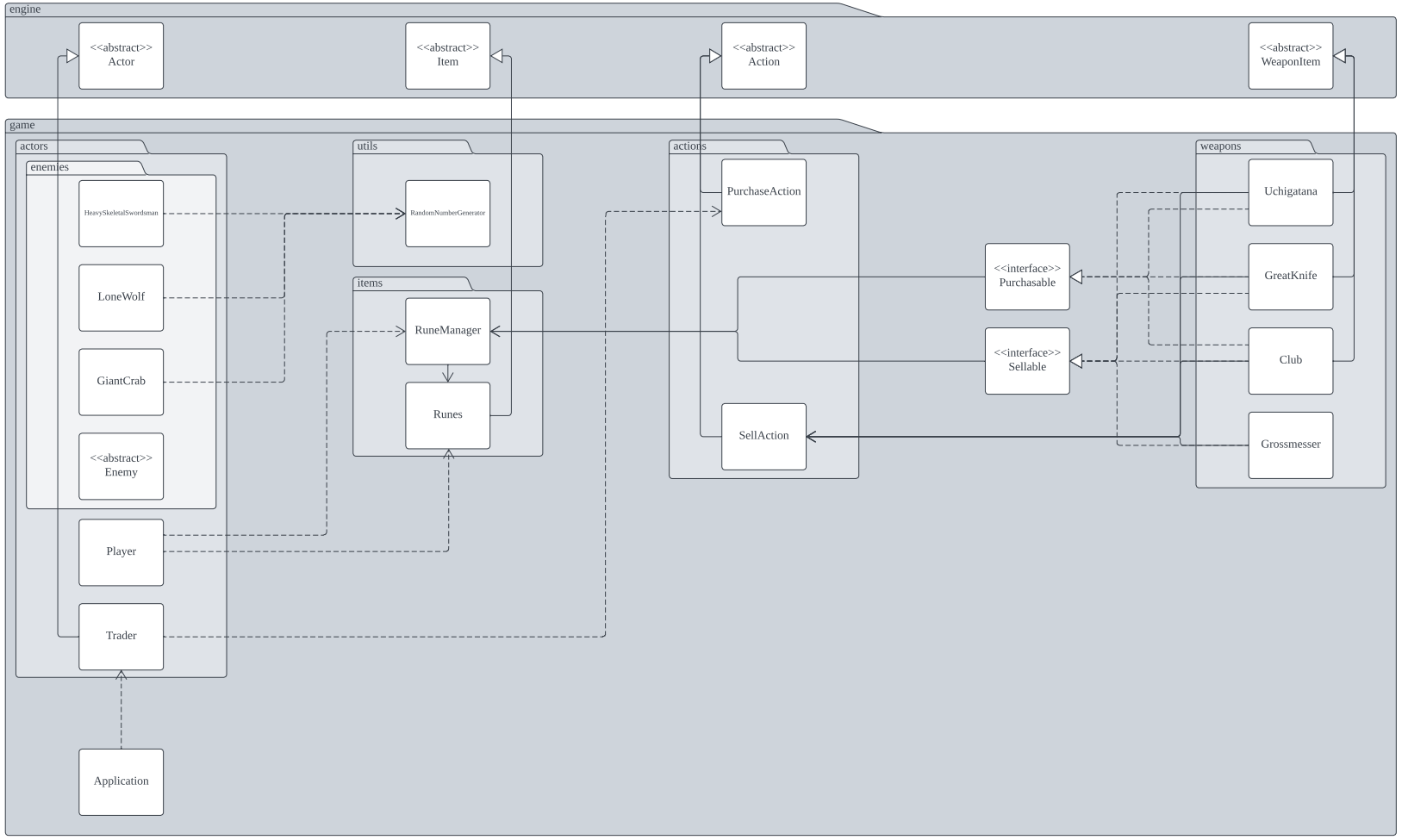
The diagram represents the second requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring. We have two design goals for this requirement: implementing runes and implementing the trader and trading actions.

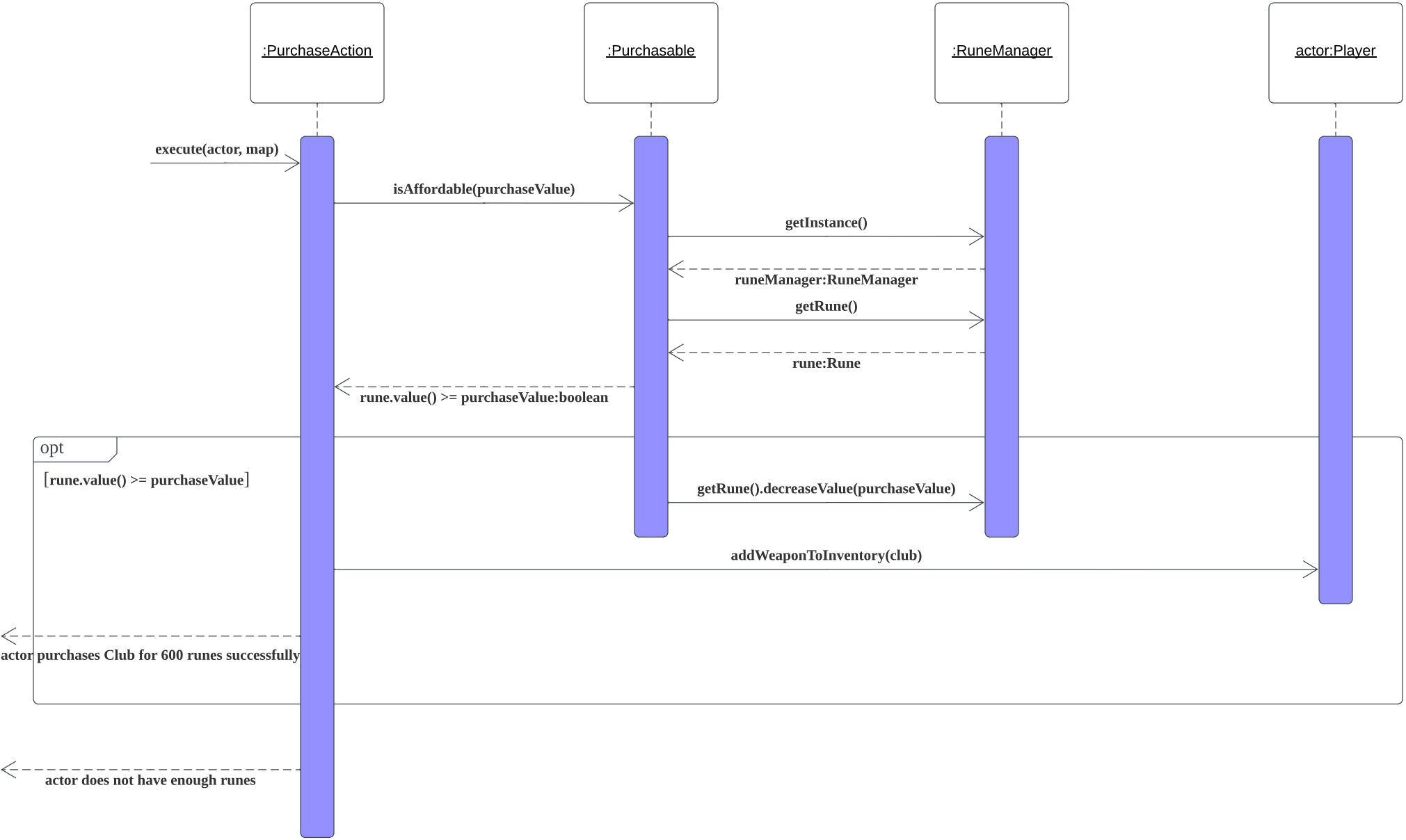
We intend to implement a different mechanism to pick up runes compared to other items. For example, when picking up runes, the runes' value would just add on to an existing runes object's value inside the player's inventory, rather than just adding new rune objects to the player's inventory everytime they pick up runes. Hence why we decided to create a new RetrieveRuneAction. Moreover, we extended RetrieveRuneAction from PickUpAction rather than PickUpItemAction because we want to avoid modifying the parent's methods completely (OCP). The same applies for DropRuneAction.

Therefore, we introduced seperate classes for purchasing and selling each individual weapon rather than a single class for purchasing/selling all kinds of weapon (e.g. PurchaseWeaponAction/SellWeaponAction). We also did this to avoid having the PurchaseAction and SellAction classes from having too many responsibilities/god class (SRP), but at the cost of repeating code.

If more purchasable weapons were to be added into the game, the weapon would extend the WeaponItem class and implement the Purchasable interface. If it is sellable, then it wiould also implement the Sellable interface. They would also need to add the purchase actions to the Trader class and sell actions to the weapon's tick function.

We then also package all the classes and interfaces into their respective package for organization, modularity, and encapsulation.


engine
<<abstract>>
Actor
<<abstract>>
Item
<<abstract>>
Action
<<abstract>>
WeaponItem
game
actors
enemies
HeavySkeletalSwordsman
LoneWolf
GiantCrab
<<abstract>>
Enemy
Player
Trader
utils
RandomNumberGenerator
items
RuneManager
Runes
actions
PurchaseAction
SellAction
<<interface>>
Purchasable
<<interface>>
Sellable
weapons
Uchigatana
GreatKnife
Club
Grossmesser
Application


:PurchaseAction
:Purchasable
:RuneManager
actor:Player
execute(actor, map)
isAffordable(purchaseValue)
getInstance()
runeManager:RuneManager
getRune()
rune:Rune
rune.value() >= purchaseValue:boolean
opt
[rune.value() >= purchaseValue]
getRune().decreaseValue(purchaseValue)
addWeaponToInventory(club)
actor purchases Club for 600 runes successfully
actor does not have enough runes

REQ 3 Design Rationale

The diagram represents the third requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring.

Previously
Individual classes of enemies (LoneWolf, GiantCrab, etc) is implemented by the Resettable interface, we used the classes individually instead of the Enemy abstract class because currently all the enemies are Resettable (can despawn/respawn when reset) but in future additions of enemy may be not Resettable.

FlaskOfCrimsonTears extends from the Item abstract class, because it has common attributes and methods with other item classes. So, it is logical to use abstraction to avoid repetitions (DRY).

ConsumeAction and RestAction classes extended the Action abstract class. Since all action classes have similar attributes and methods, it is logical to extend ConsumeAction and RestAction from Action abstract class to avoid code repetitions (DRY).

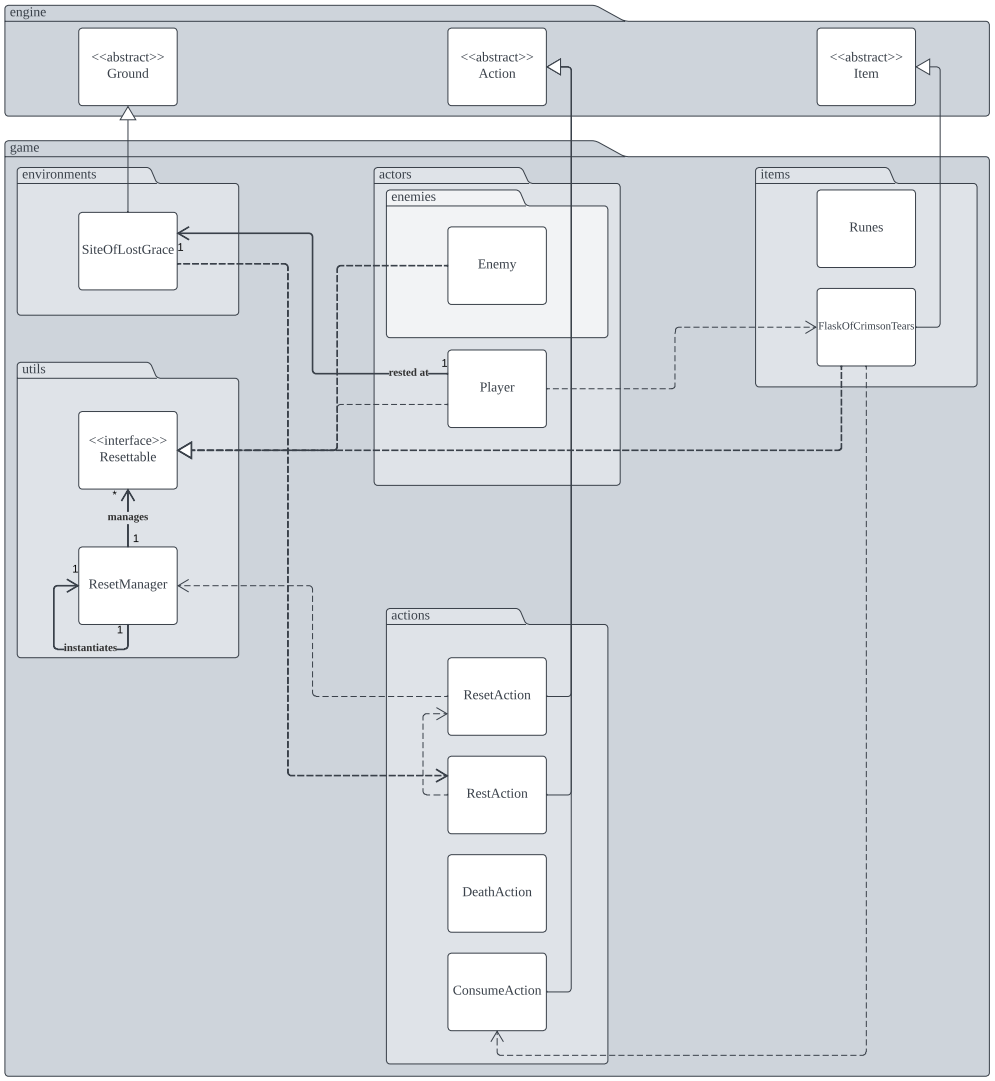
SiteOfLostGrace class extends from Ground abstract class, because it is a new environment class and all environment classes have similar attributes and methods, so to avoid repetition we use abstraction (DRY).

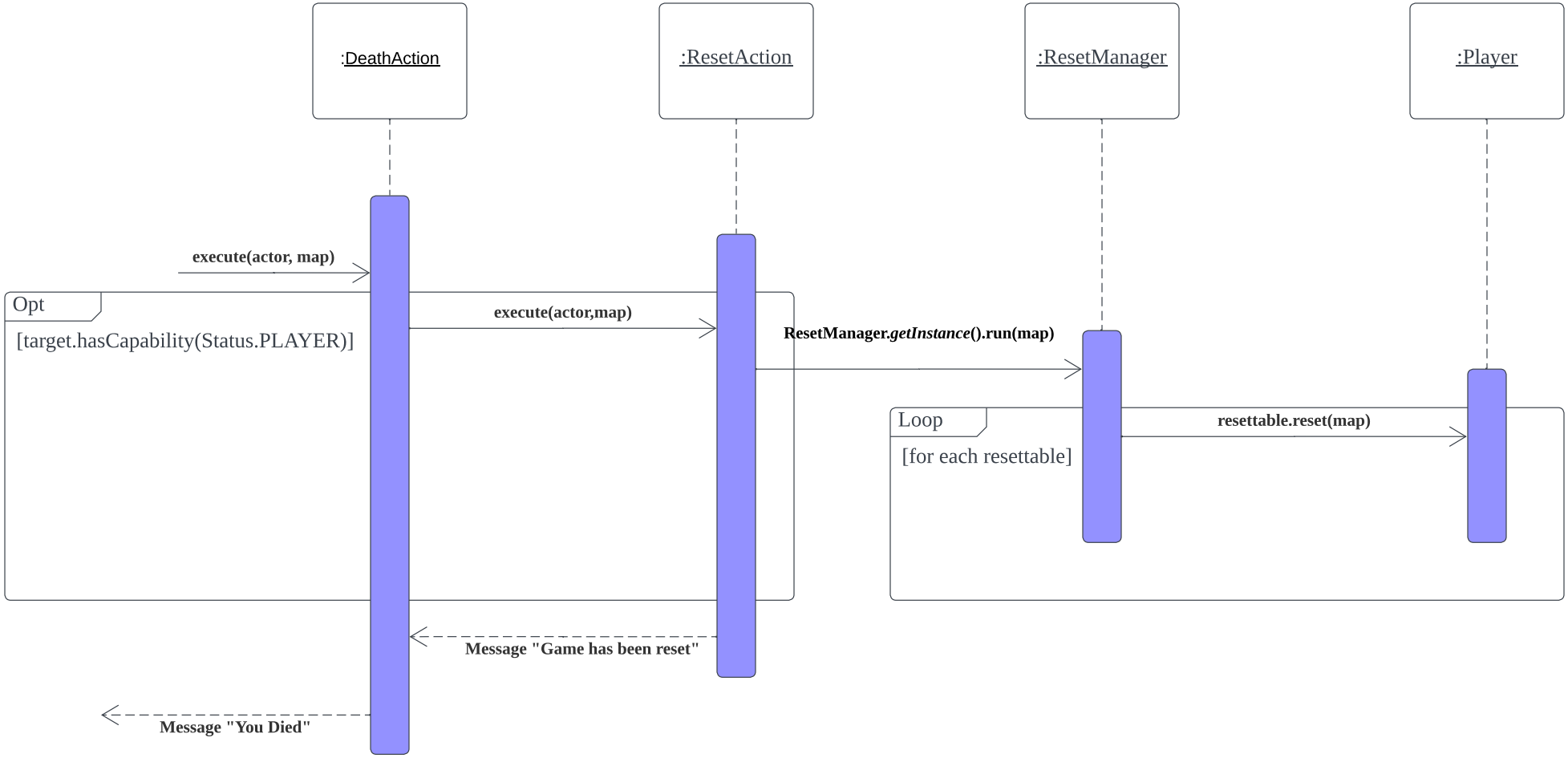
The ResetManager manages and depend on the Resettable interface rather than directly to the item or actor classes, this way any classes that implements the Resettable interface can be used with the ResetManager (DIP).

New Changes
Due to Pile of Bones not being an actor, we have decided to just implement the Enemy abstract class because all the enemies are Resettable, this is to avoid code repetition for the Reset method for each enemies.

FlaskOfCrimsonTears implements a Resettable while also extending from the Item abstract class, this change was applied so that in the future we can add items as Resettable as well instead of Resetting it all on the Player class.

Added ResetAction to initialize the reset, instead of letting all of the reset happening on the ResetManager, so that in the future new Behaviours and other classes can utilize the Reset.


engine
<<abstract>>
Ground
<<abstract>>
Action
<<abstract>>
Item
game
environments
SiteOfLostGrace
1
actors
enemies
Enemy
items
Runes
FlaskOfCrimsonTears
rested at
1
Player
utils
<<interface>>
Resettable
*
manages
1
1
ResetManager
1
instantiates
actions
ResetAction
RestAction
DeathAction
ConsumeAction


:DeathAction
:ResetAction
:ResetManager
:Player
execute(actor, map)
Opt
[target.hasCapability(Status.PLAYER)]
execute(actor,map)
ResetManager.getInstance().run(map)
Loop
[for each resettable]
resettable.reset(map)
Message "Game has been reset"
Message "You Died"

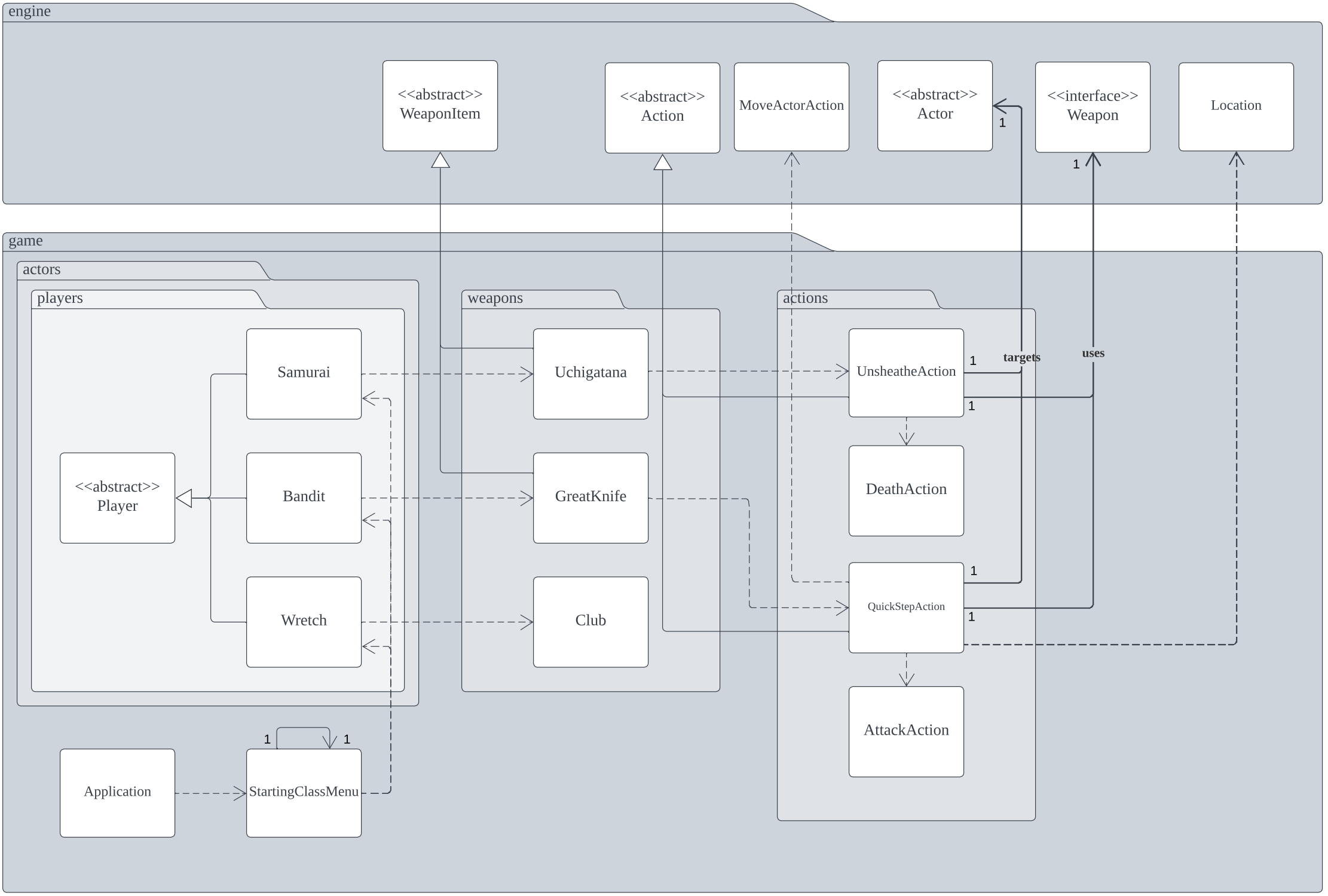
REQ 4 Design Rationale

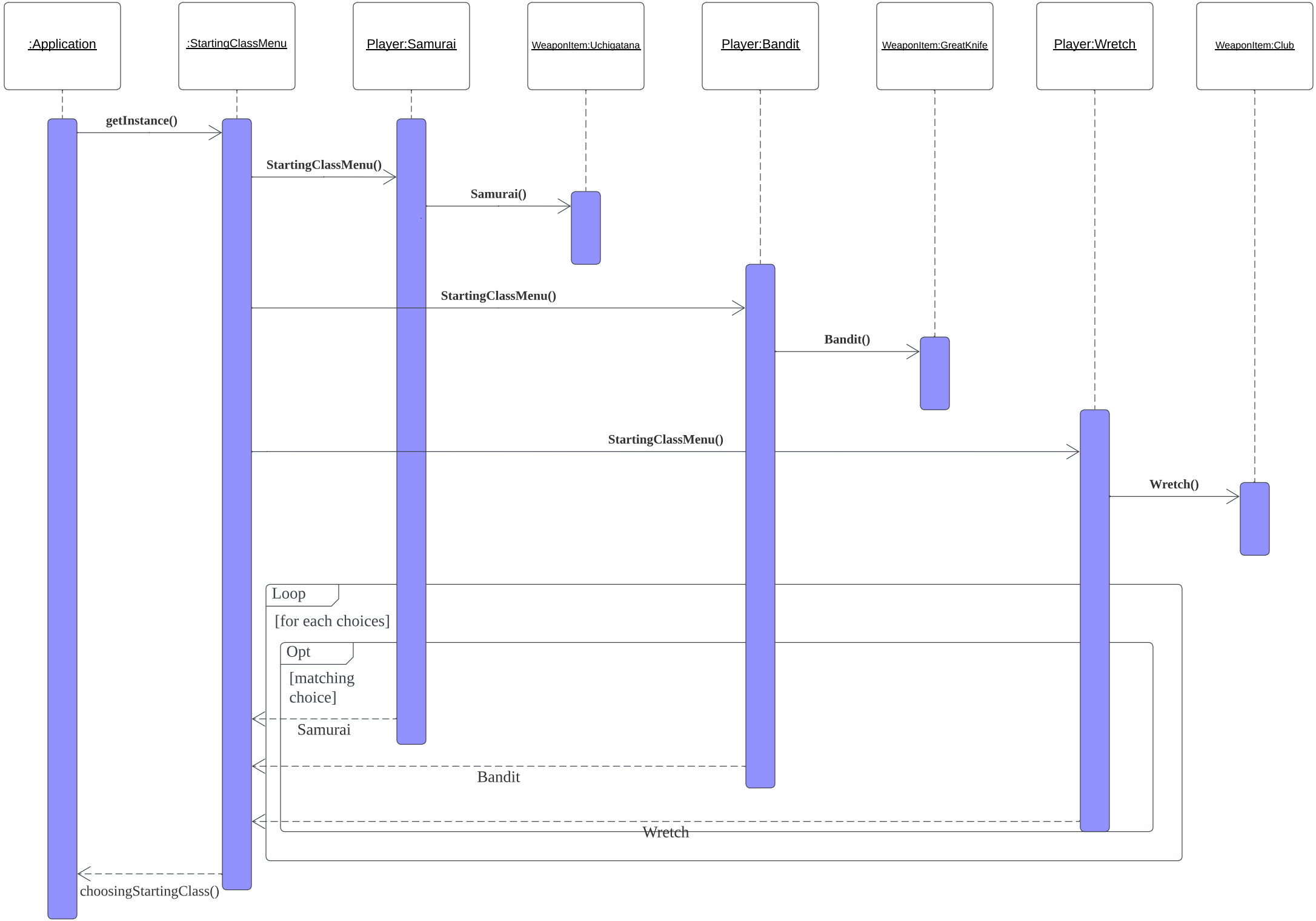
The diagram represents the fourth requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring. We have two design goals for this requirement: classes/combat archetypes and weapons.

Each combat archetypes have similar attributes, only differences are hitpoints and starting weapon. Since every combat archetype should have the same attributes and methods that a Player has, the Player class is changed into an abstract class, and each combat archetype class extends from the Player abstract class to avoid code repetition (DRY). By using this way, when adding a new combat archetype, we do not have to repeat coding the attributes and methods used by that class.

Uchigatana and GreatKnife extended the WeaponItem abstract class. Since all weapons share some common attributes and methods, it is logical to extend Uchigatana and GreatKnife from WeaponItem abstract class so that Uchigatana and GreatKnife have all the attributes and methods that a weapon should have without code repetition (DRY). This also adheres to DIP where a concrete class should not depend on another concrete class but instead on abstractions. For example, when we want to add GreatKnife to the inventory of Samurai, we do not have to create an add weapon method to add that specific weapon type which also violates OCP. Instead, we only need one add weapon method that is able to add any type of weapon to inventory.

DoubleDamageAction and EvadeAttackAction are special skills of Uchigatana and GreatKnife, therefore DoubleDamageAction have dependency to Uchigatana and EvadeAttackAction have dependency to GreatKnife. DoubleDamageAction and EvadeAttackAction extends the Action abstract class since they share common attributes and methods. This makes it so that DoubleDamageAction and EvadeAttackAction can be implemented without code repetition (DRY).


engine
<<abstract>>
WeaponItem
<<abstract>>
Action
MoveActorAction
<<abstract>>
Actor
<<interface>>
Weapon
Location
1
1
game
actors
players
Samurai
<<abstract>>
Player
Bandit
Wretch
weapons
Uchigatana
GreatKnife
Club
actions
UnsheatheAction
DeathAction
QuickStepAction
AttackAction
1
1
1
1
targets
uses
Application
StartingClassMenu
1
1


:Application
:StartingClassMenu
Player:Samurai
WeaponItem:Uchigatana
Player:Bandit
WeaponItem:GreatKnife
Player:Wretch
WeaponItem:Club
getInstance()
StartingClassMenu()
Samurai()
StartingClassMenu()
Bandit()
StartingClassMenu()
Wretch()
Loop
[for each choices]
Opt
[matching choice]
Samurai
Bandit
Wretch
choosingStartingClass()

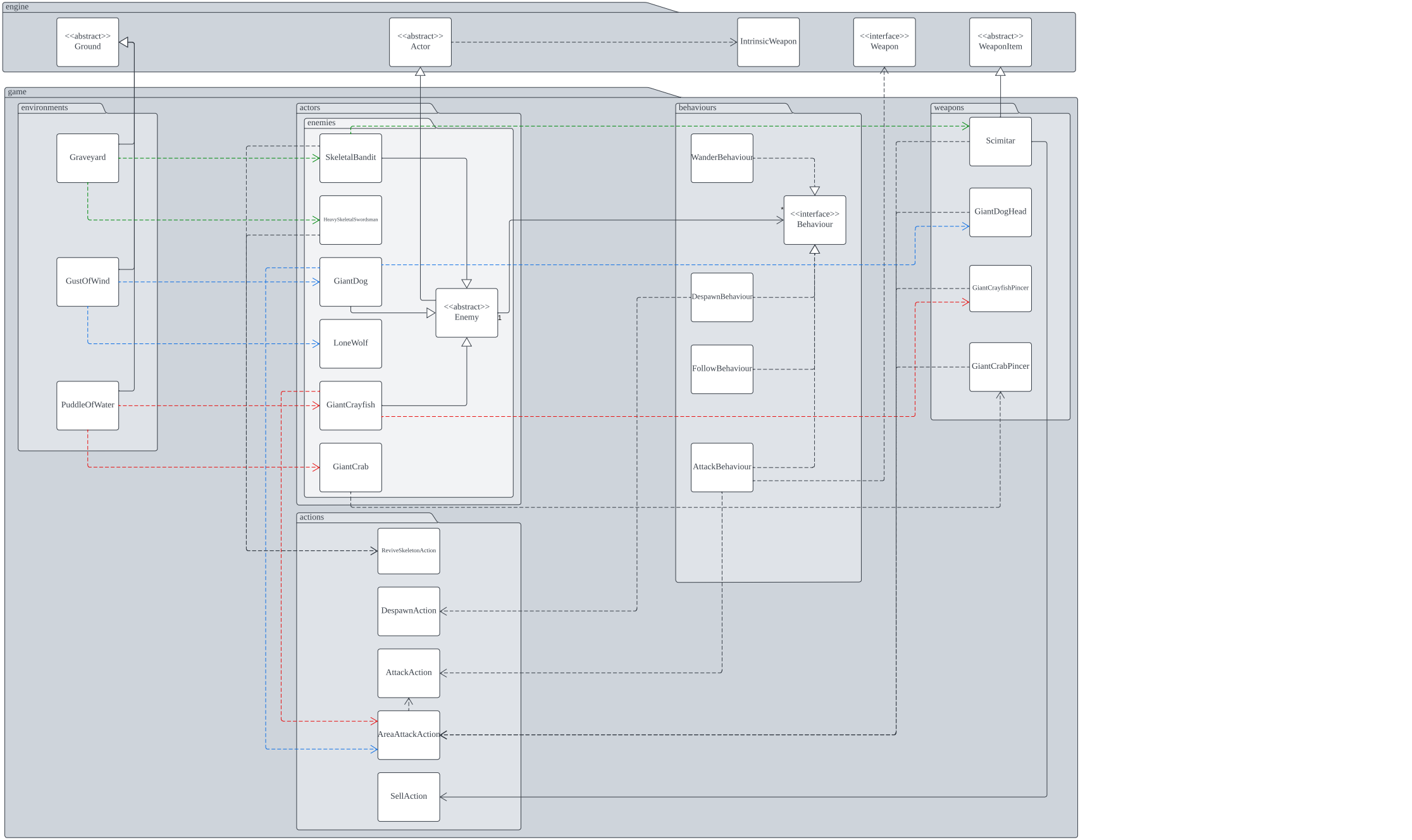
REQ 5 Design Rationale

The diagram represents the fifth requirement/part of an object-oriented system for a text-based “rogue-like” game inspired by Elden Ring. We have two design goals for this requirement: implementing more enemies and weapons.

SkeletalBandit, GiantDog and GiantCrawfish extends Enemy abstract class since all enemies have some common attributes and methods. This makes SkeletalBandit, GiantDog and GiantCrawfish have all the attributes and methods that an enemy should have while avoiding code repetition (DRY).

Skeleton, Beast and Crustacean are implemented as interfaces to avoid multi-level inheritance from Enemy class. If we were to create an enemy like SkeletalDog, we can have it implement Skeleton and Beast interfaces, which will not work if we have them as abstract classes. This way of implementing also makes the system more extensible, new operations could be easily added to the system by adding more interfaces since each of the current interfaces is separated by its own purpose (ISP).

Since all weapons share some common attributes and methods, Scimitar extended the WeaponItem abstract class so that Scimitar have all the attributes and methods that a weapon should have without code repetition (DRY).


engine
<<abstract>>
Ground
<<abstract>>
Actor
IntrinsicWeapon
<<interface>>
Weapon
<<abstract>>
WeaponItem
game
environments
Graveyard
GustOfWind
PuddleOfWater
actors
enemies
SkeletalBandit
HeavySkeletalSwordsman
GiantDog
LoneWolf
GiantCrayfish
GiantCrab
<<abstract>>
Enemy
1
actions
ReviveSkeletonAction
DespawnAction
AttackAction
AreaAttackAction
SellAction
behaviours
WanderBehaviour
<<interface>>
Behaviour
*
DespawnBehaviour
FollowBehaviour
AttackBehaviour
weapons
Scimitar
GiantDogHead
GiantCrayfishPincer
GiantCrabPincer