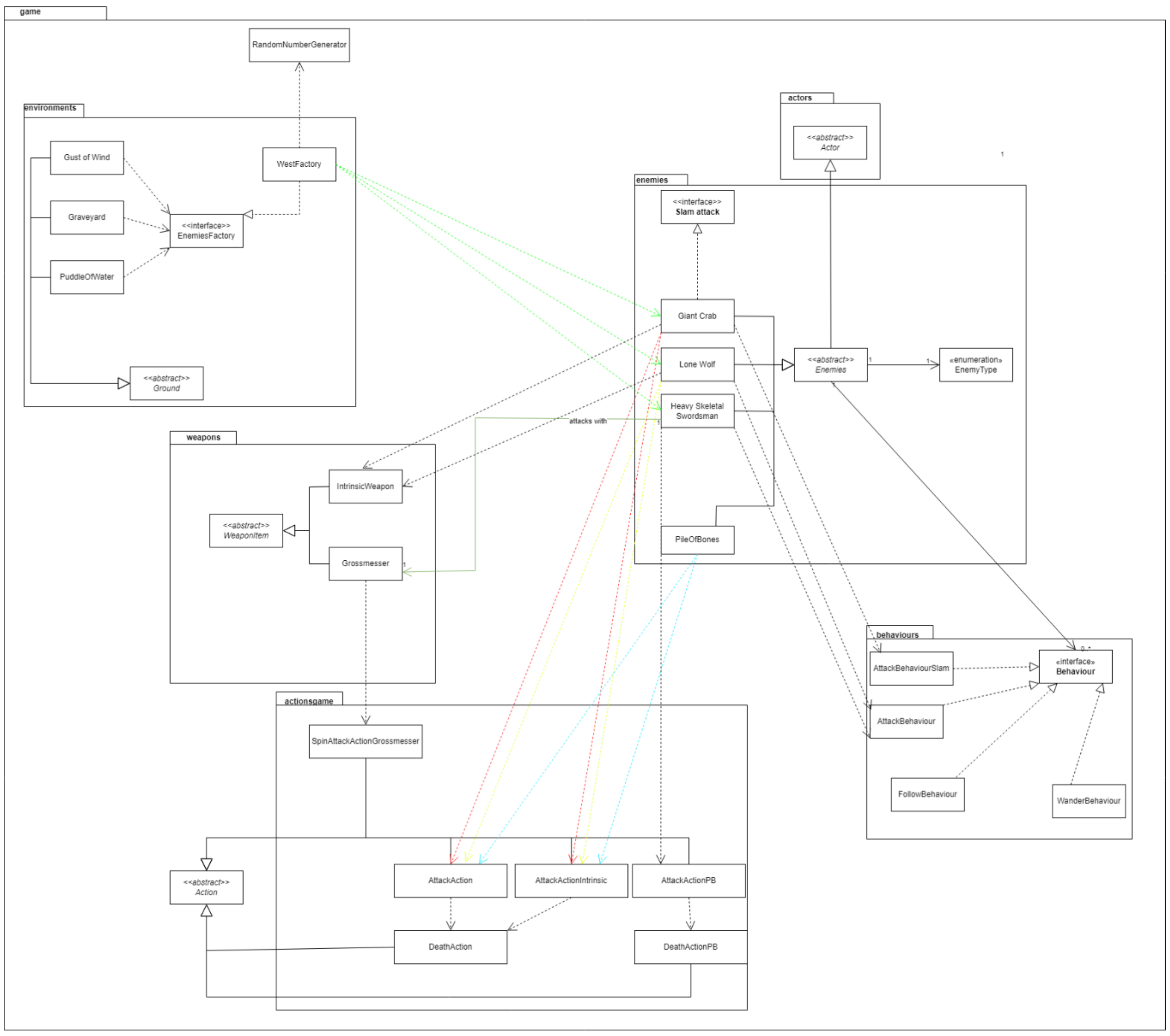
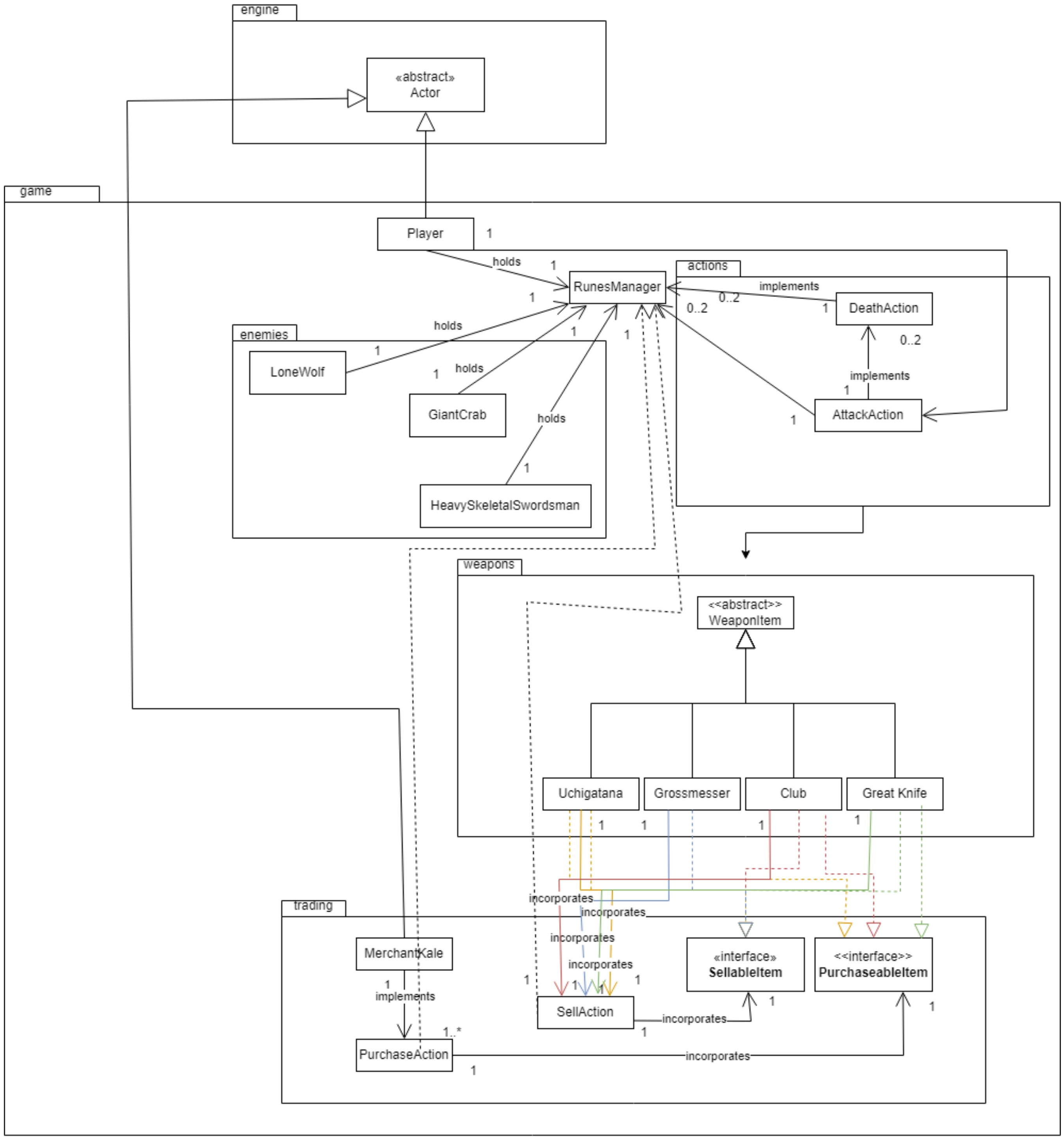
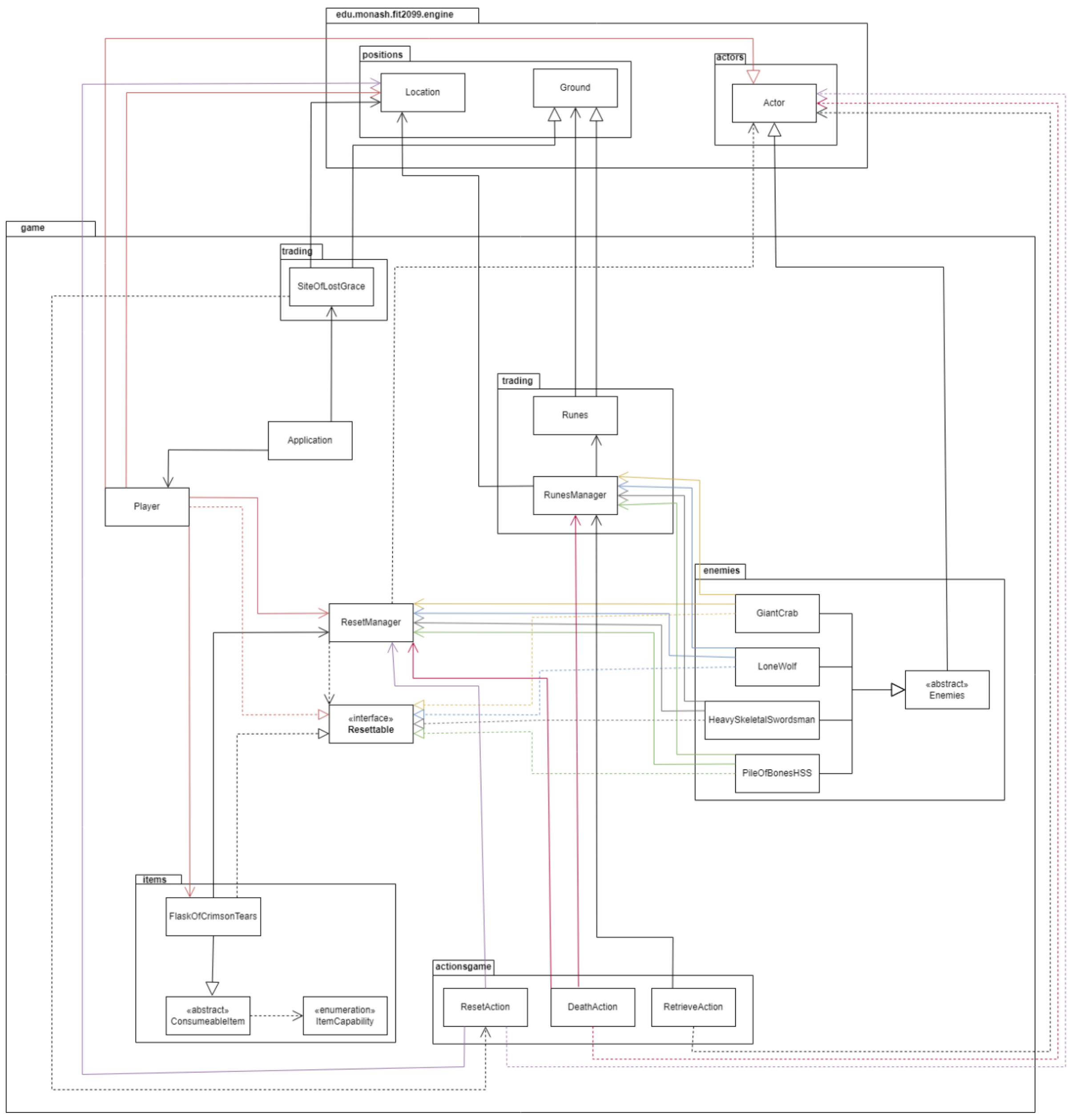
UML Requirement 1



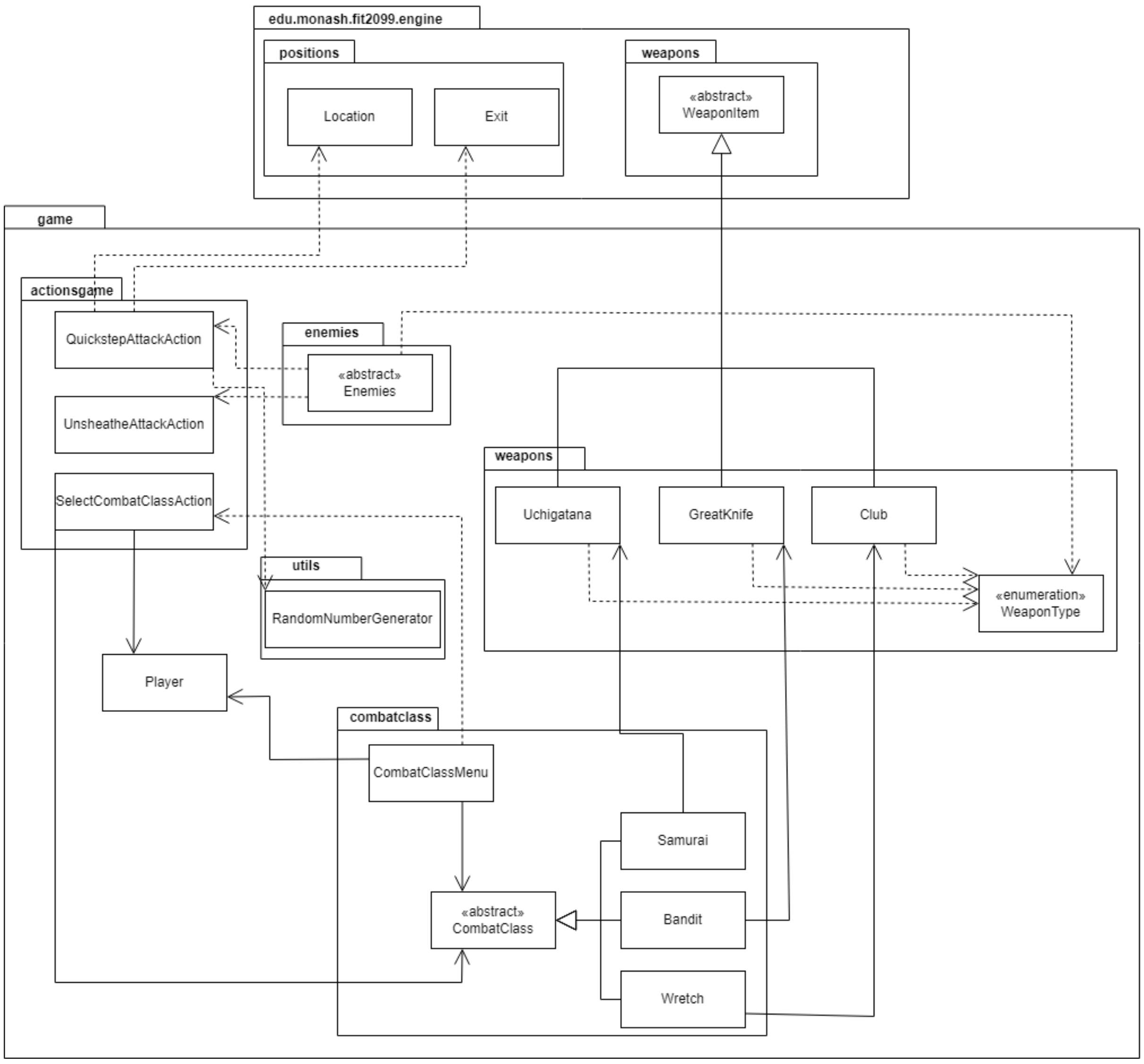
UML Requirement 2



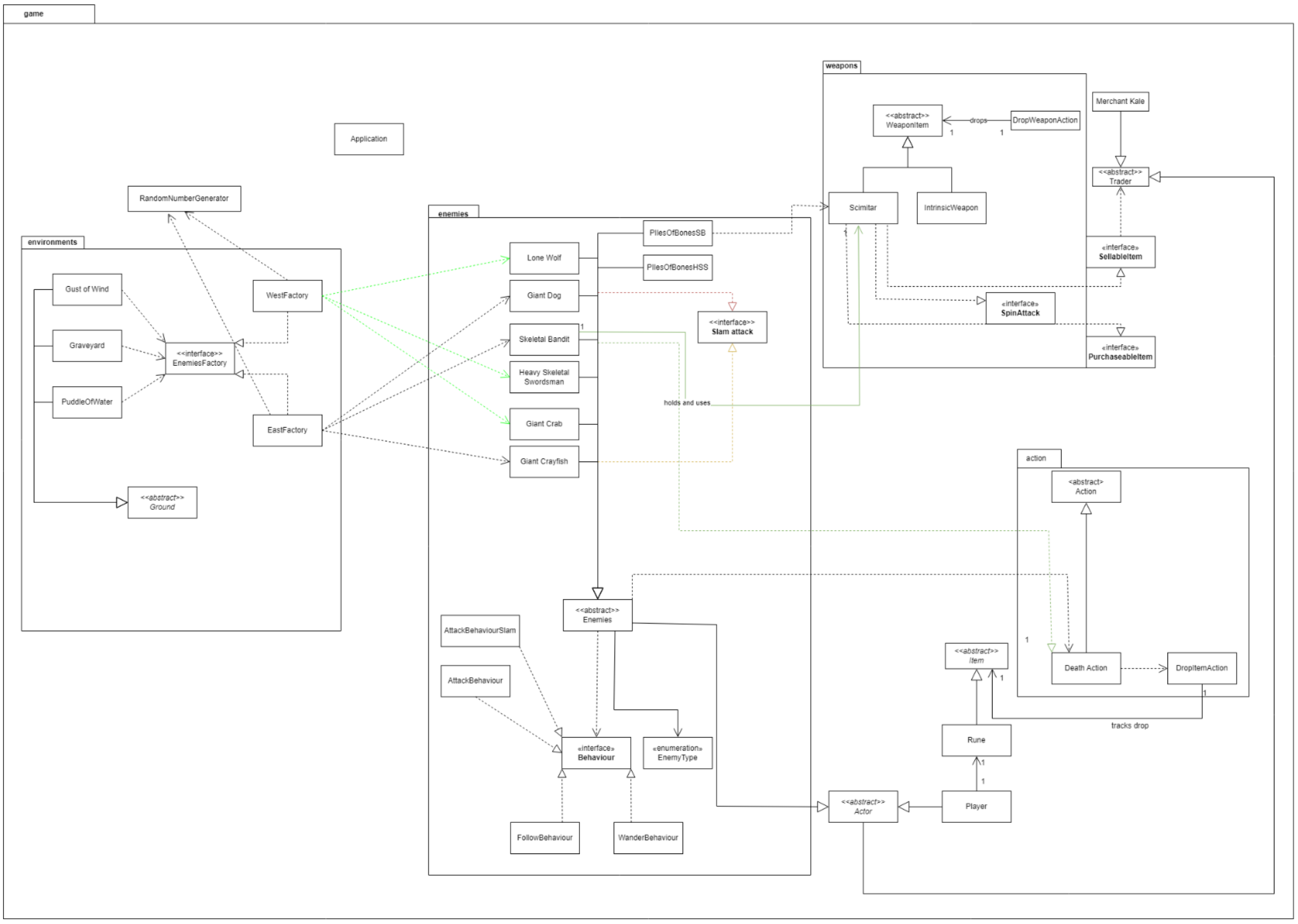
UML Requirement 3



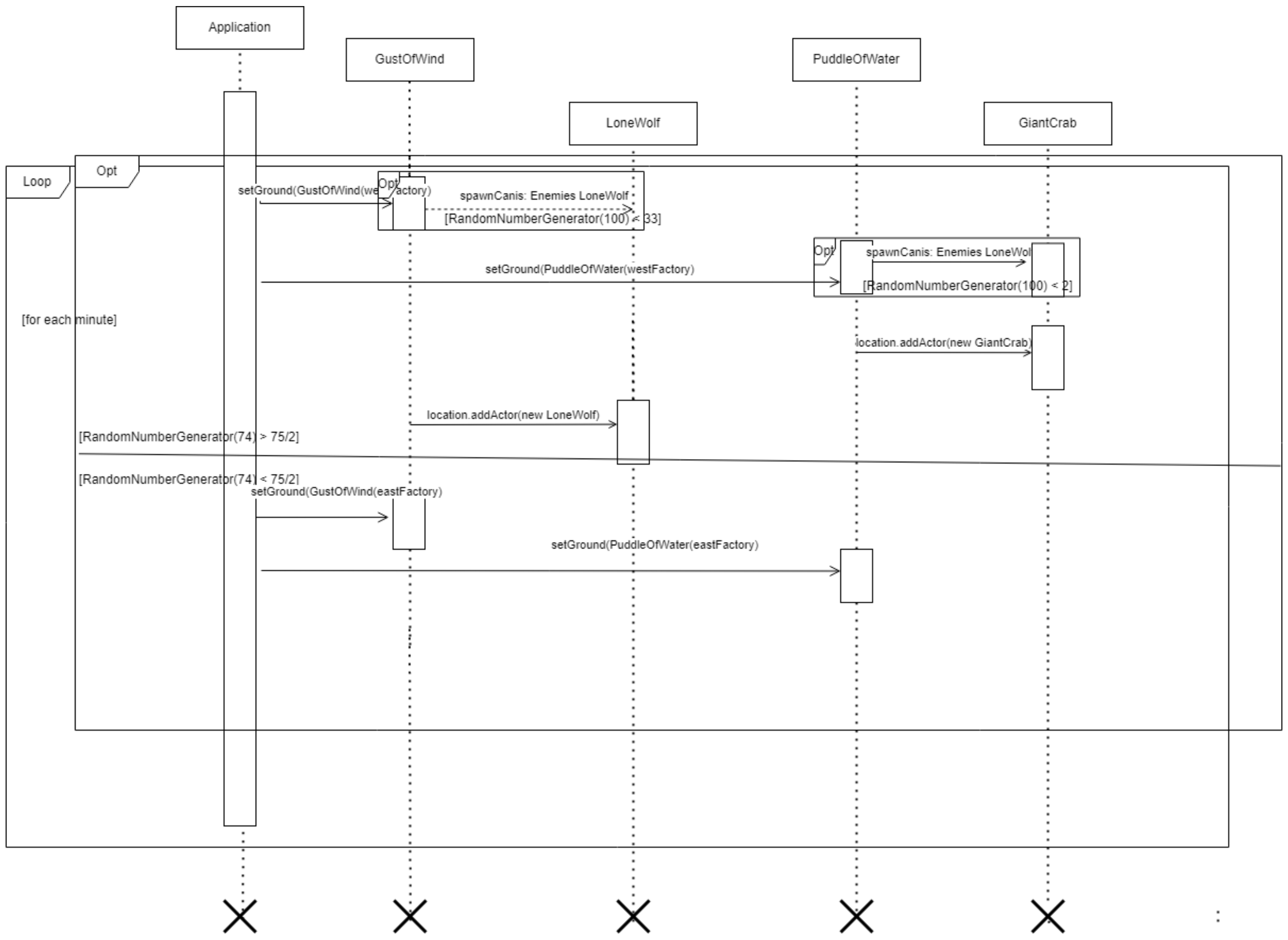
UML Requirement 4



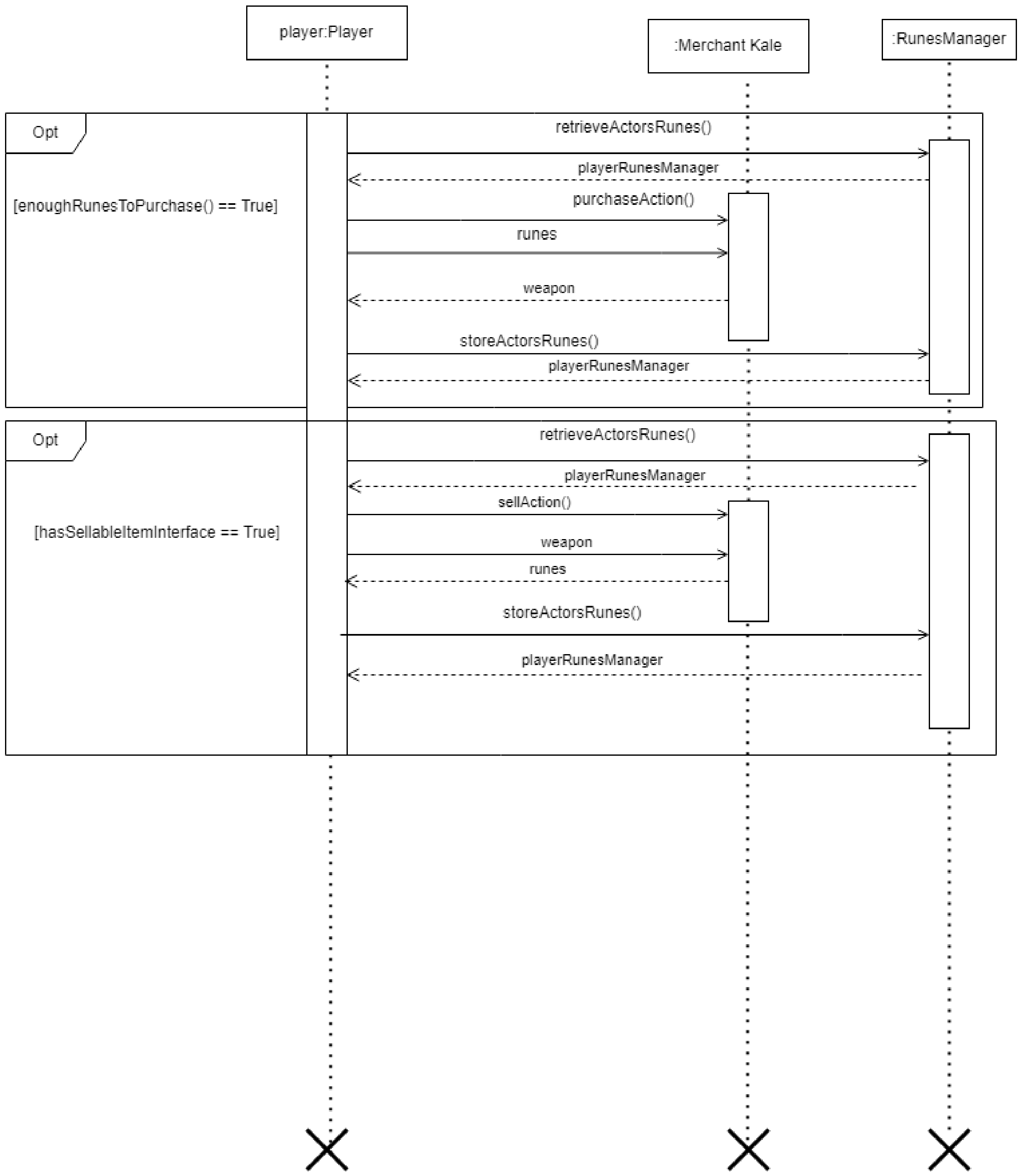
UML Requirement 5



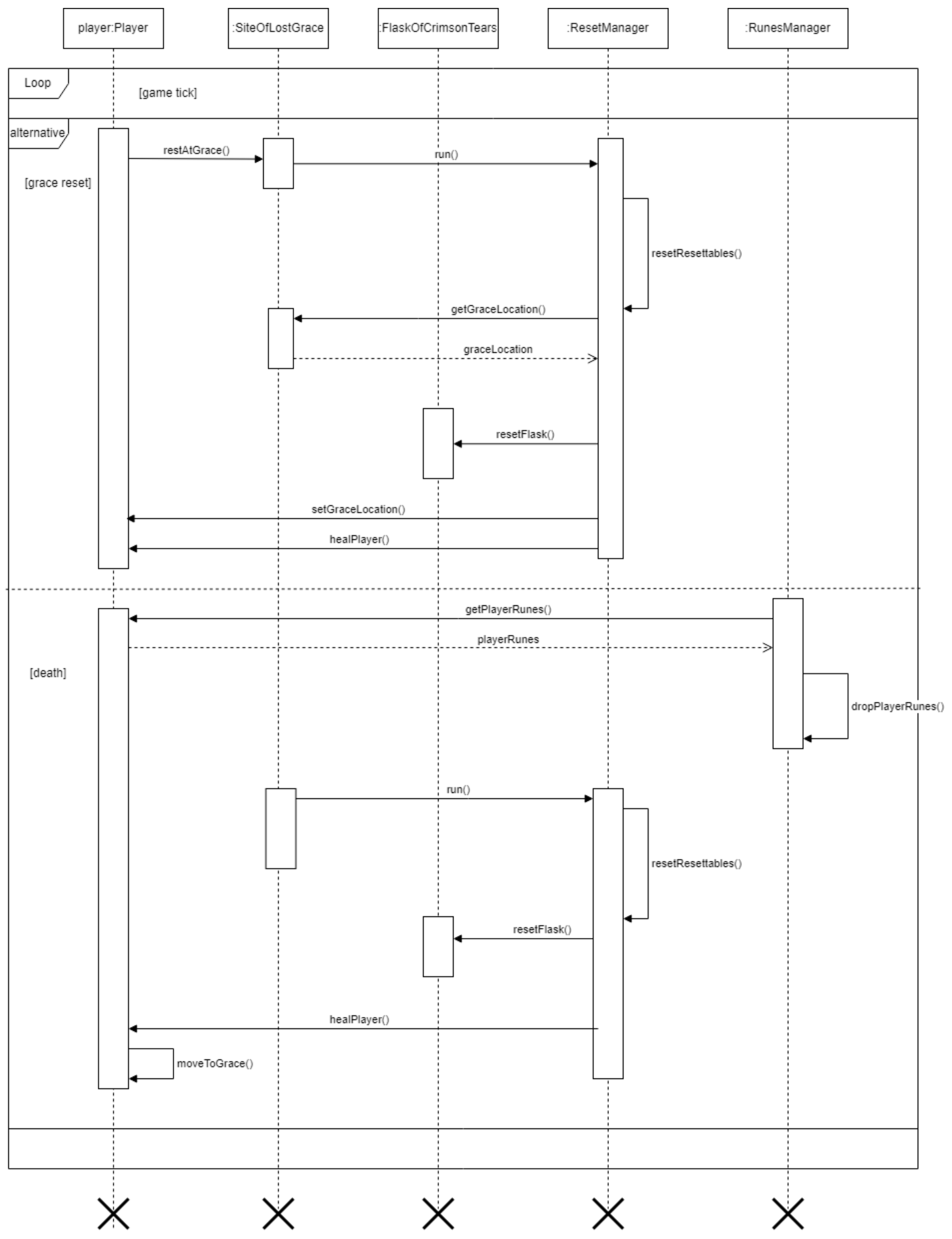
Sequence Diagram Requirement 1


Application
GustOfWind
LoneWolf
PuddleOfWater
GiantCrab
Loop
Opt
[for each minute]
setGround(GustOfWind(westFactory)
Opt
spawnCanis: Enemies LoneWolf
[RandomNumberGenerator(100) < 33]
setGround(PuddleOfWater(westFactory)
Opt
spawnCanis: Enemies LoneWolf
[RandomNumberGenerator(100) < 2]
location.addActor(new GiantCrab)
location.addActor(new LoneWolf)
[RandomNumberGenerator(74) > 75/2]
[RandomNumberGenerator(74) < 75/2]
setGround(GustOfWind(eastFactory)
setGround(PuddleOfWater(eastFactory)

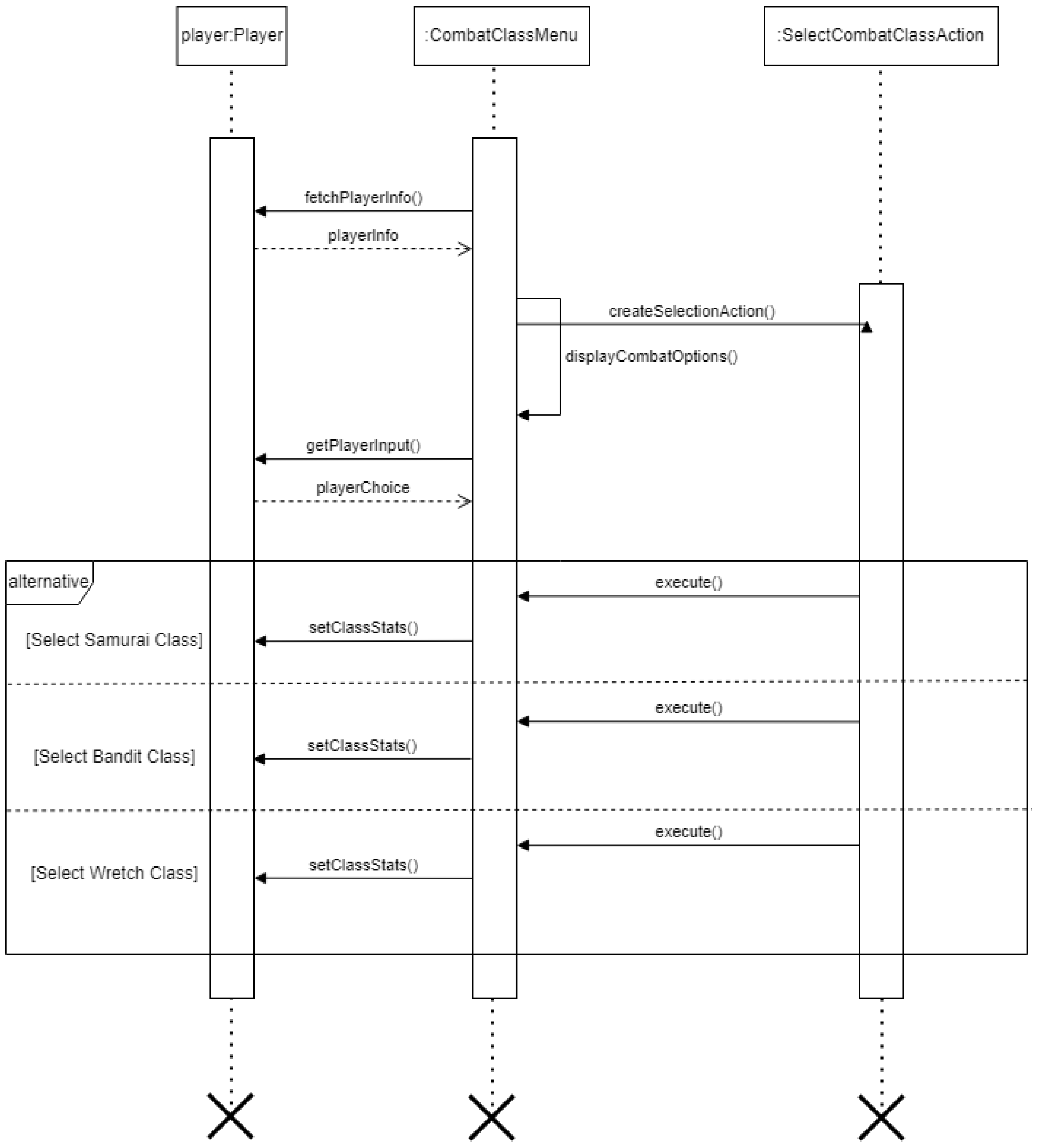
Sequence Diagram Requirement 2



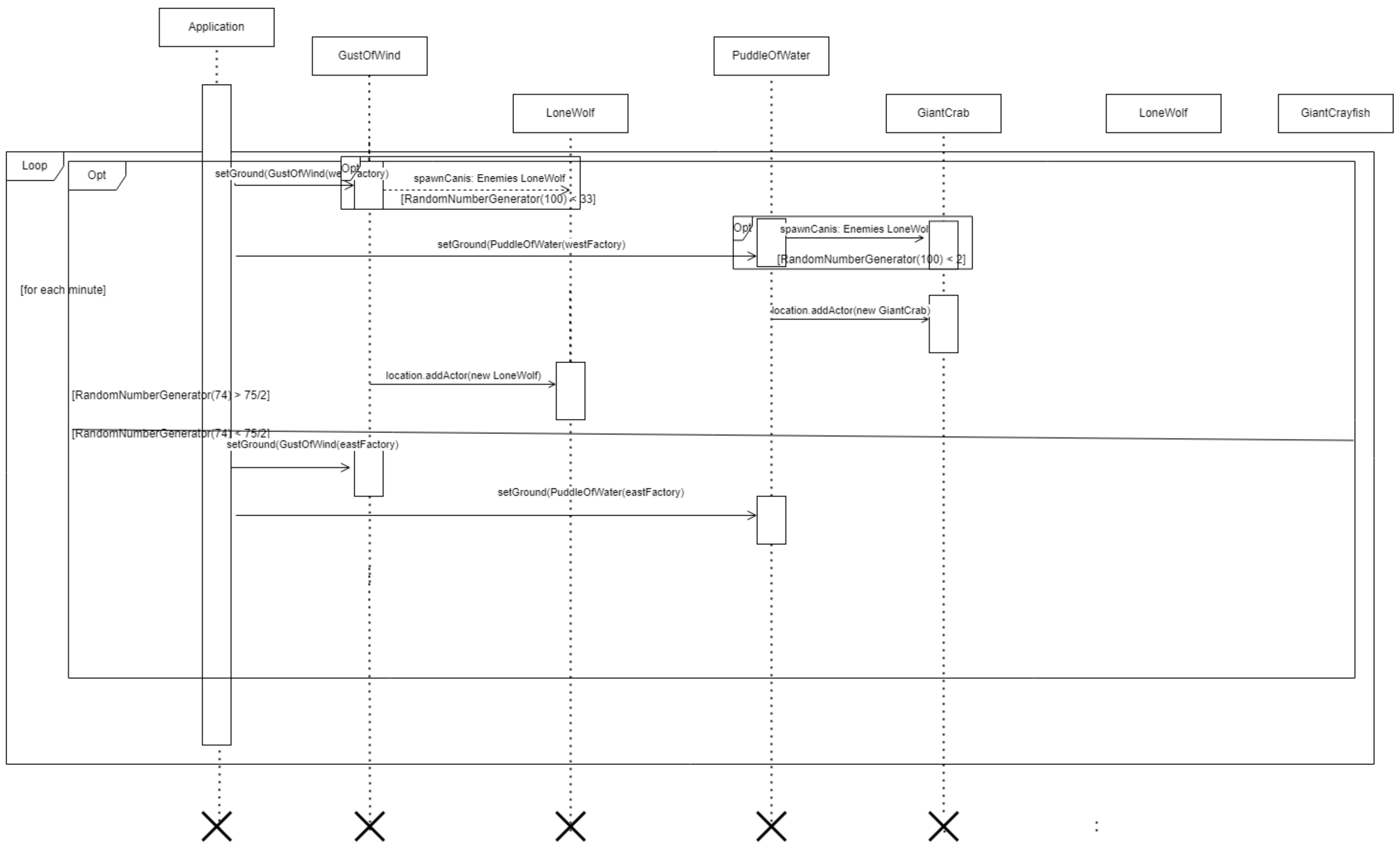
Sequence Diagram Requirement 3



Sequence Diagram Requirement 4



Sequence Diagram Requirement 5


Application
GustOfWind
PuddleOfWater
LoneWolf
GiantCrab
LoneWolf
GiantCrayfish
Loop
Opt
setGround(GustOfWind(westFactory)
Opt
spawnCanis: Enemies LoneWolf
[RandomNumberGenerator(100) < 33]
Opt
spawnCanis: Enemies LoneWolf
setGround(PuddleOfWater(westFactory)
[RandomNumberGenerator(100) < 2]
[for each minute]
location.addActor(new GiantCrab)
location.addActor(new LoneWolf)
[RandomNumberGenerator(74) > 75/2]
[RandomNumberGenerator(74) < 75/2]
setGround(GustOfWind(eastFactory)
setGround(PuddleOfWater(eastFactory)

Requirement 1

The UML diagram represents a system that will manage different game environments and enemies. It utilises various in-built game engine classes as well as custom classes to achieve an object-oriented design that meets the design requirements.

The abstract ground class acts as a base class for the different environments; Puddle of Water, Gust of Wind and Graveyard, further, it is an abstract class as its instantiation is never required. These child classes are inhabited by specific enemies. This was implemented in this way such that if we were to add a new environment, it can simply inherit from the abstract class making it easy to extend.

The enemy classes; Giant Crab, Lone Wolf and Heavy Skeletal Swordsman, are extended in front of the abstract Enemies class, to keep in line with DRY principles. The abstract Enemies class is an extension of the abstract Actors class, this is given in the game engine. Enemies extend this class as the Actors class is used by the game engine for various functions such as movement and game tick which are in-built. The Enemies class has an association with the enumeration EnemyType, where it is an attribute of the class. This association has been utilised to account for different enemy types. Regardless of the fact that there are only 3 different enemies, each with a unique type, this implementation accounts for future requirements where there are different enemies of the same type.

The Enemies class also has an association with the Behaviour Interface, which is implemented in the FollowBehaviour and WanderBehaviour classes. This allows for these classes to be utilised within the Enemies class, of which the required enemy behaviour can be used within the game loop. Further, the use of a behaviour interface means that specific class types don’t have to be considered, rather, as long as the class implements the Behaviour interface it can be used by the Enemies class. The SpawnAction class is a dependency relation with the Enemies class, allowing enemies to be spawned during the game loop. There is also a dependency between the SpawnAction class and the RandomNumberGenerator class, this allows for the instantiation of enemies at random times.

The child classes that extend the Enemies class include Pile of Bones, which is only spawned if the health of a Heavy Skeletal Swordsman drops to zero. As such there is also a dependency present with the Heavy Skeletal Swordsman. The Giant Crab class will also implement the SlamAttack interface.

Regarding enemy attacks, each enemy has some form of weapon, either IntrinsicWeapon or Grossmesser, which extends the abstract WeaponItem class. Grossmessed class implements the SpinAttack interface as it is a unique attack for the Grossmesser.

New implementations

Implemented an enemies factory interface, which includes an EastFactory and WestFactory, for future requirements, as this will allow different sides to have different enemies of the same type. Thus, GustOfWind, Graveyard and PuddleOfWater will have dependencies on the interface. Whereas the EastFactory and WestFactory will implement them, and if you pass in East or West to the biome class, you will get a different enemy depending on the side. This is good for when we extend the game if we ever decide to create a North or South side, as this will make it very easy to spawn specific types of enemies.

Furthermore, rather than doing an interface for the spin attack, we opted for an action to conduct it to make it in line with our other implementations of AttackAction.

On the other hand, for slam attacks for enemies that can use area-of-effect attacks such as Giant Crab, we have opted for a SlamAttackBehaviour, which would have a 50% chance of adding AttackAction or SlamAttackAction. This is beneficial for future implementations because when we add more enemies that can use slam attacks, we only have to add this behaviour to the enemies, which enables them to slam using their own intrinsic weapon damage and accuracy.

Requirement 2

The diagram represents how the system will incorporate the universal currency, runes, traders and tradeable items to interact with its existing classes.

The enemy classes LoneWolf, GiantCrab and HeavySkeletalSwordsman each have an association to the RunesManager class as each drop different random amounts of runes when they die from the player. The RunesManager is a singleton class that holds all the runes by any actor. By doing so, we uphold the Open Closed principle, so that any new actors added to the game will only extend the code but not modified it. However, it breaches the Single Responsibility principle, since all actors rely on this class to access their own runes. This acts as a god class for all the runes in the game. This also avoids the need for casting when

The action classes AttackAction and DeathAction also have an association to the RunesManager class. AttackAction is designed in this way to ensure that only those actors that are killed by the player drop runes, preventing runes from dropping when one enemy kills another enemy. This also upholds the Single Responsibility Principle as it ensures each different action class only contains one action responsibility.

The PurchaseAction and SellAction have a dependency to the RunesManager class, as trading between player and Merchant Kale requires runes in order for the transfer to be successful.

The sellable weapons, Uchigatana, Grossmesser, Club and Great Knife weapon classes all have an association to the SellAction class. This is juxtaposed to having Merchant Kale have an association to the SellAction class, since logically Merchant Kale does not know what weapons are in the player’s inventory at any given time. Instead, whenever the Merchant Kale is nearby, the player will be able to sell each possible sellable weapon to the trader. This promotes the Single Responsibility Principle as Merchant Kale only holds the purchaseable weapons.

SellableItem and PurchaseableItem are implemented as interfaces. This allows us to avoid multi-level inheritance from the WeaponItem class, since not all weapons can be purchased from Kale, reinforcing the Interface Segregation Principle. This also prevents the abstract WeaponItem class from becoming a god class and enables more interfaces to be easily added since all the current interfaces are separated by their own purpose.

Pros
Avoids having each actor hold their runes, which avoids casting in the SellAction and PurchaseAction classes

Avoid having Merchant Kale responsible for both selling and purchasing of weapons completely.

Cons
All transfer, gain or dropping of runes rely solely on the RunesManager class

Requirement 3

The UML demonstrates the game reset and player death actions in a more verbose manner.

ResetManager and RunesManager classes have been used to form a central set of classes to deal with game reset and runes management, aligning with the Single Responsibility Principle. This ensures that runes and resetting occur and are managed when they are needed, and not at any other point during execution. Its also important to note that the ResetManager has a dependency with the Actor class, which are extended in the Player and Enemies classes. This ensures that when handling the reset for all these classes we only need to reference the Actor class, and perform necessary actions with that abstract class. This ensures that reusable code is written to deal with any Actor instance, rather than catering for a variety of classes that require resetting, upholding the Liskov substitution principle.

A Resettable interface is also implemented for the classes that require resetting, this is also referenced as a dependency in the ResetManager. This ensures that the ResetManager is the only class that is responsible for the actual resetting actions. Further, only the required classes implement the Resettable interface, aligning with the Interface Segregation principle.

Both the ResetManager and the Player classes have an association with the FlaskOfCrimsonTears. FlaskOfCrimsonTears itself is extended from the ConsumeableItem abstract class, although there is only the one consumable item to consider currently, for future implementations, it will allow for simple expansion. An ItemCapability enumeration is also used to define what the consumable item is actually capable of, this can be used inline with the Engine capability sets to define what actions can be performed, however, this is only required for consideration for future implementations. As the Flask is associated with both ResetManager and Player, the Flask will be a singleton, meaning the single instance will be used for all instantiations. This allows for easy access and modification of attributes, supporting the Open/Closed principle.
Although the bulk of the reset action occurs in the ResetManager, the SIteOfLostGrace and ResetAction are still required. By having a dependency from SiteOfLostGrace to ResetAction and an association from ResetAction to ResetManager, there is a direct dependency that allows for the reset to occur centrally. This support the Open/Closed principle. There is a similar association between DeathAction and ResetManager, to allow for resetting of the game when the player dies, again supporting the Open/Closed principle.

The Runes class is used to allow for dropping runes upon player death, it extends the Ground class. There is an assocation with the RunesManager, and as there is an association between the RunesManager and the Player classes, the attributes required for instantiation of the Runes class can simply be called from the RunesManager. Again the central control of the runes for all actors in the game is managed here. In this way any number of actors implemented in the future can have runes managed by this system.

Requirement 4

The design diagram considers the implementation of combat classes for the game. This is done by having the player class form an association with the abstract CombatClass. The different combat archetypes; Samurai, Bandit and Wretch, extend the abstract CombatClass. In this way, a specific combat class can be associated with the Player class. The Player Class itself extends the Actor abstract class, given in the game engine, which allows for the various functions of the Actor class to still be available while implementing Player functionality.

The different combat classes use different weapons, and as such different weapon classes are required. An abstract WeaponItem class is used as a base, where different weapons are thereby extended to create the Grossmesser, Uchigatana, Great Knife and Club classes. It's also important to note that the WeaponItem class has an association with the enumeration WeaponType, which can easily define what the type of weapon is.

The combat classes Samurai, Bandit and Wretch all extend the CombatClass class. This upholds the Open Closed Principle as it ensures that additional combat classes added in the future will not modify the code, but can be extended to incorporate these new features.

The QuickstepAttackAction has a dependency with the Location class, as Actor position must be known in order to perform the ability.

The Enemies class have a dependency to the two weapon attack abilities, QuickStepAttackAction and UnsheatheAttackAction. This ensures that if the player uses that particular weapon, then they will be able to attack the enemy with the unique ability. The benefit of having the abilities as actions rather than interfaces enables the weapons to specificly target the particular enemy with the ability. However, this violates the Open Closed Principle, as the inclusion of more unique weapon abilities will result in the enemies having dependencies to many different abilities, if more of them are added in the future. However, this may violate the Interface Segregation Principle as no interfaces were used to specify the distinct abilities of each weapon, instead we used actions.

Pros
More combat classes can be added in the future

Cons
Without using interfaces for each weapon ability that are implemented by each specific weapon

Requirement 5

The UML diagram for requirement 5 explains how we will incorporate another three enemies into the game, whilst also dividing the map into two halves namely west and east, of which enemies can spawn in certain halves. Furthermore, the diagram builds on a new weapon, the Scimitar which can be purchased and sold to the merchant.

Implemented an enemies factory interface, which includes an EastFactory and WestFactory, for future requirements, as this will allow different sides to have different enemies of the same type. Thus, GustOfWind, Graveyard and PuddleOfWater will have dependencies on the interface. Whereas the EastFactory and WestFactory will implement them, and if you pass in East or West to the biome class, you will get a different enemy depending on the side. As such EastFactory will spawn SkeletalBandit, GiantDog, and GiantCrayfish in their respective environments on the East, and then the WestFactory will spawn HeavySkeletalSwordsman, LoneWolf and GiantCrab on the West. This is good for when we extend the game if we ever decide to create a North or South side, as this will make it very easy to spawn specific types of enemies.

In our implementations for the three new enemies, Skeletal Bandit, Giant Dog and Giant Crayfish, similar to previous diagrams we ensured that each enemy inherits the enemy class and that the enemy class will inherit the actor class. Likewise, behaviours will implement the behaviour class to form the FollowBehaviour and WanderBehaviour of each enemy. Furthermore, each environment will have an attribute to which enemies populate the biome. An advantage of this implementation is that it suffices the DRY principle in that we are not repeating attributes such as the health points of the enemies. Another advantage is the LSP principles, in that we can add more enemies and thus allows polymorphism which is easier to extend code. A disadvantage of our implementation is that every time we need to add a new enemy, the biomes must have an association with it making it slightly complicated.

Specifically, when a skeletal bandit dies, a pile of bones will spawn. We have implemented such that upon the skeletal bandit dying will call DeathAction and spawn a pile of bones. After three turns, the skeletal bandit will respawn which is shown by the dependency between the pile of bones and SpawnAction if it is not hit by the player. The pile of bones will call DeathAction and spawn a scimitar weapon on the ground. An advantage of this is that the pile of bones inherits from the Enemy class, which applies the DRY principle and thus reuses attributes and methods across subclasses. The disadvantage of this approach is that it is not very extendable in that if we were to add another object similar to a pile of bones, we would have to implement it the same way the pile of bones was which is not efficient. A way to fix this would be to

In order to implement the new weapon, the Scimitar, like previous weapons inherits the WeaponItem class. Since the skeletal bandit holds the weapon, it has an association. Implementing the scimitar's skill SpinAttack was done through an interface as this would all enable ISP as this segregates features of weapons, and if a new weapon was to be added that has spin attack as well, we could simply implement it for that new weapon, which benefits hugely in terms of simplicity.