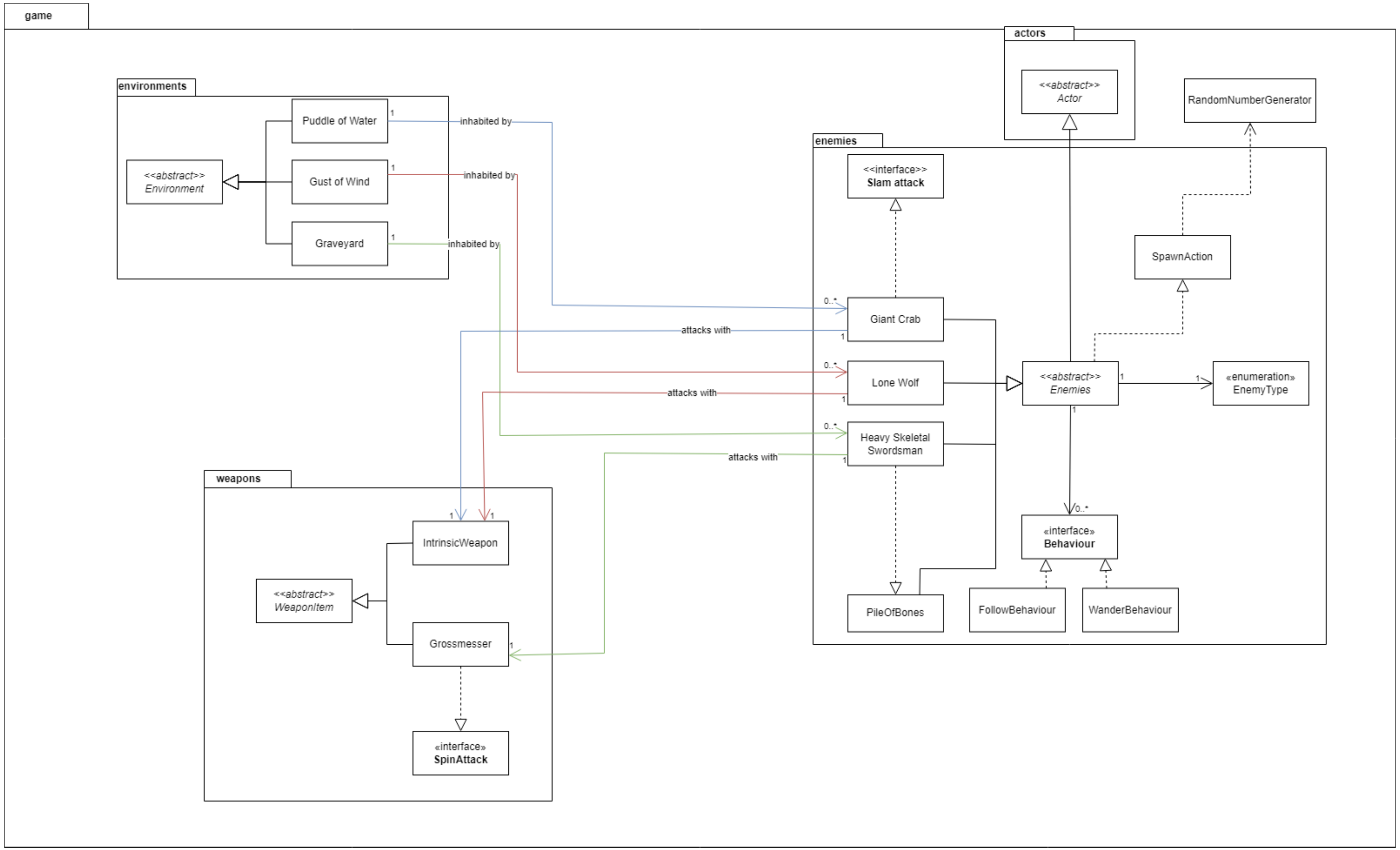
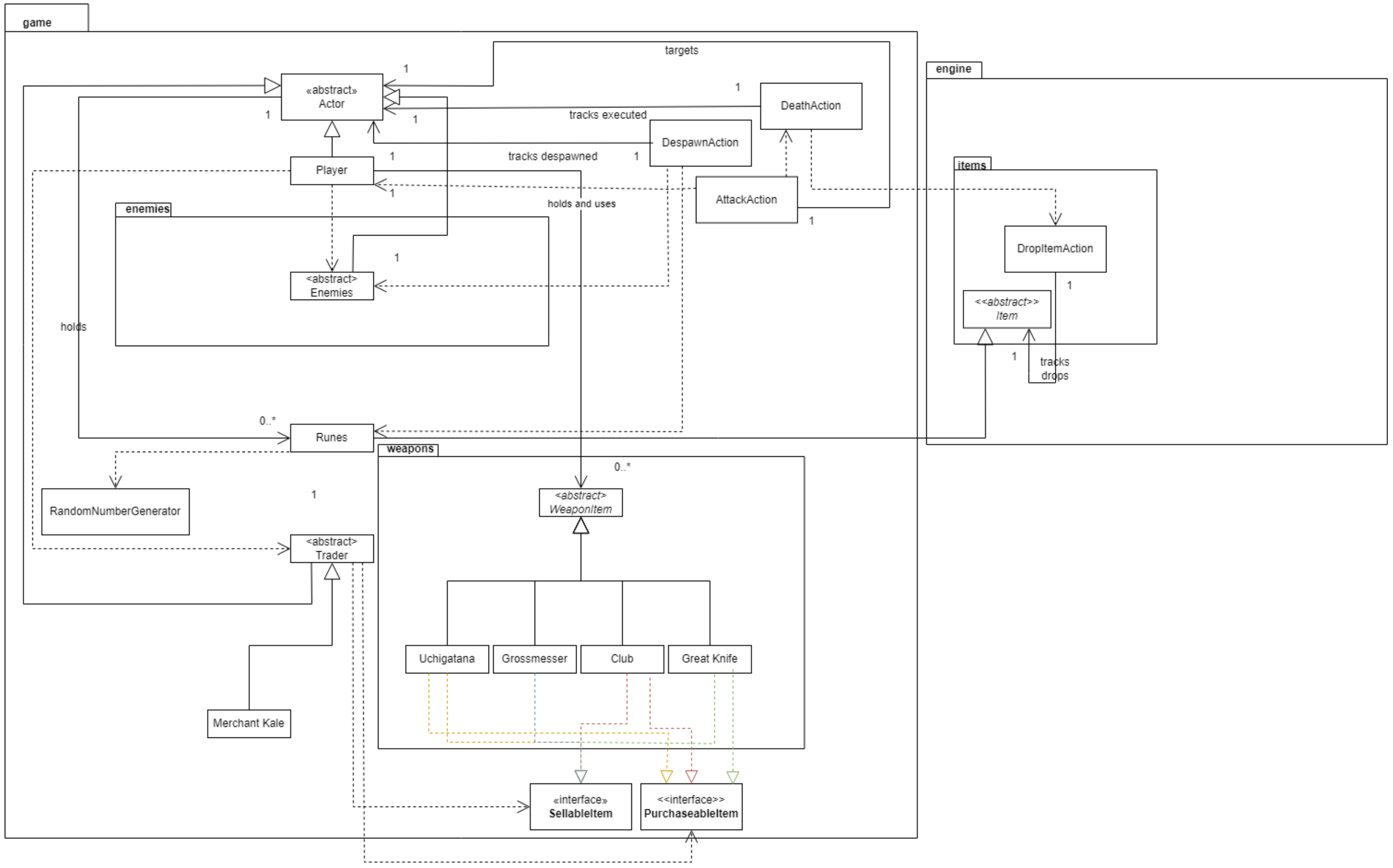
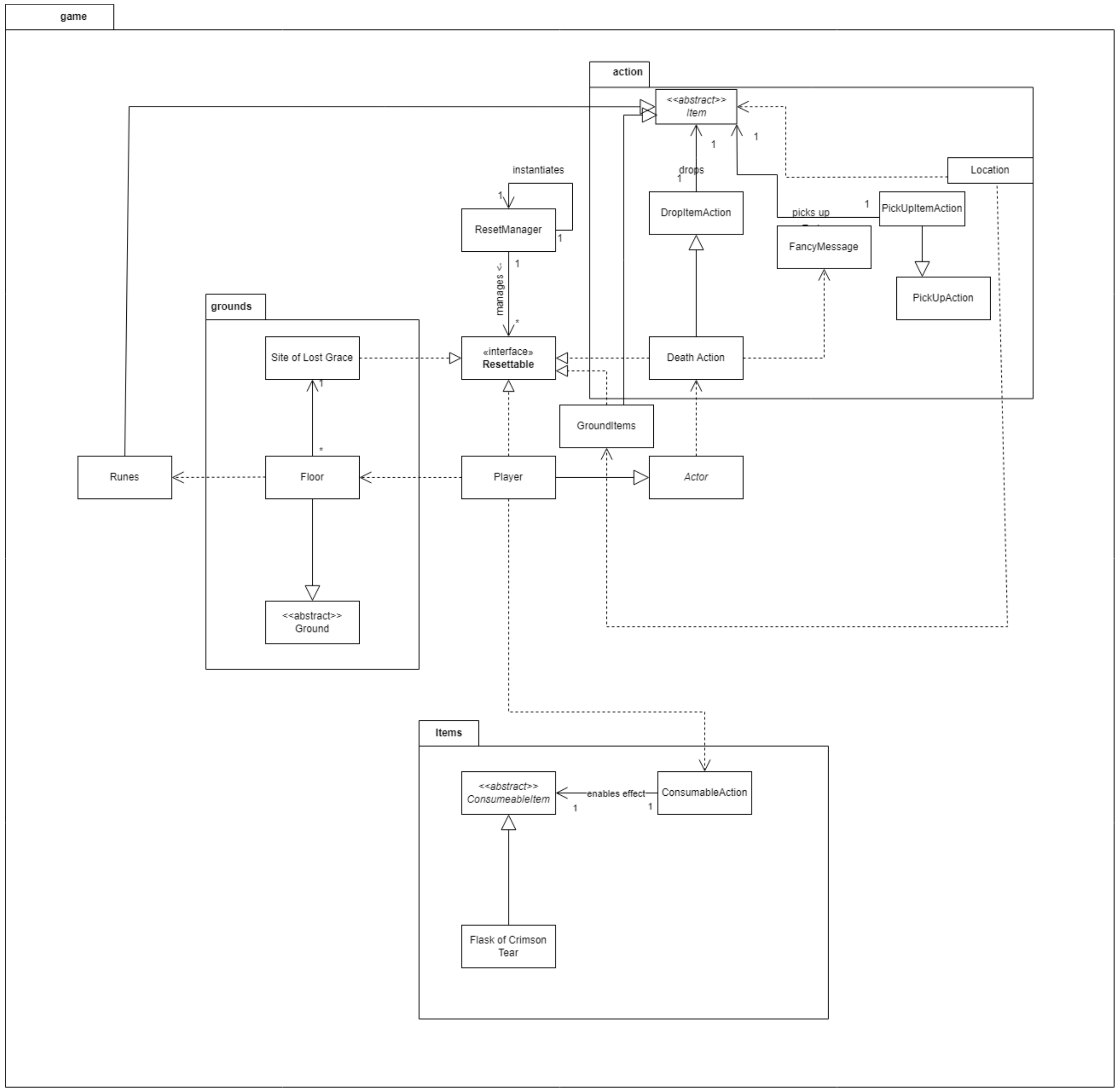
UML Diagram Requirement 1



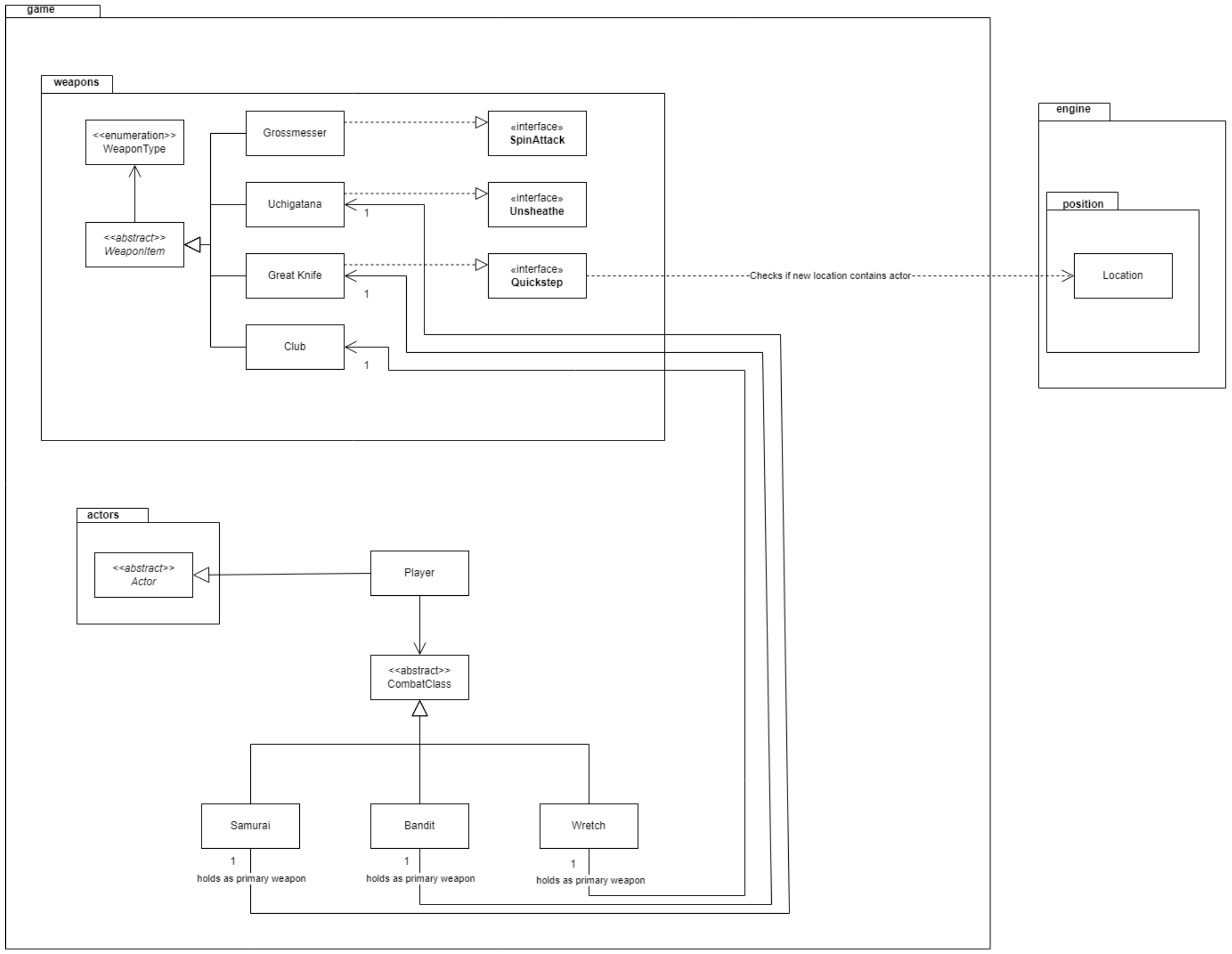
UML Diagram Requirement 2



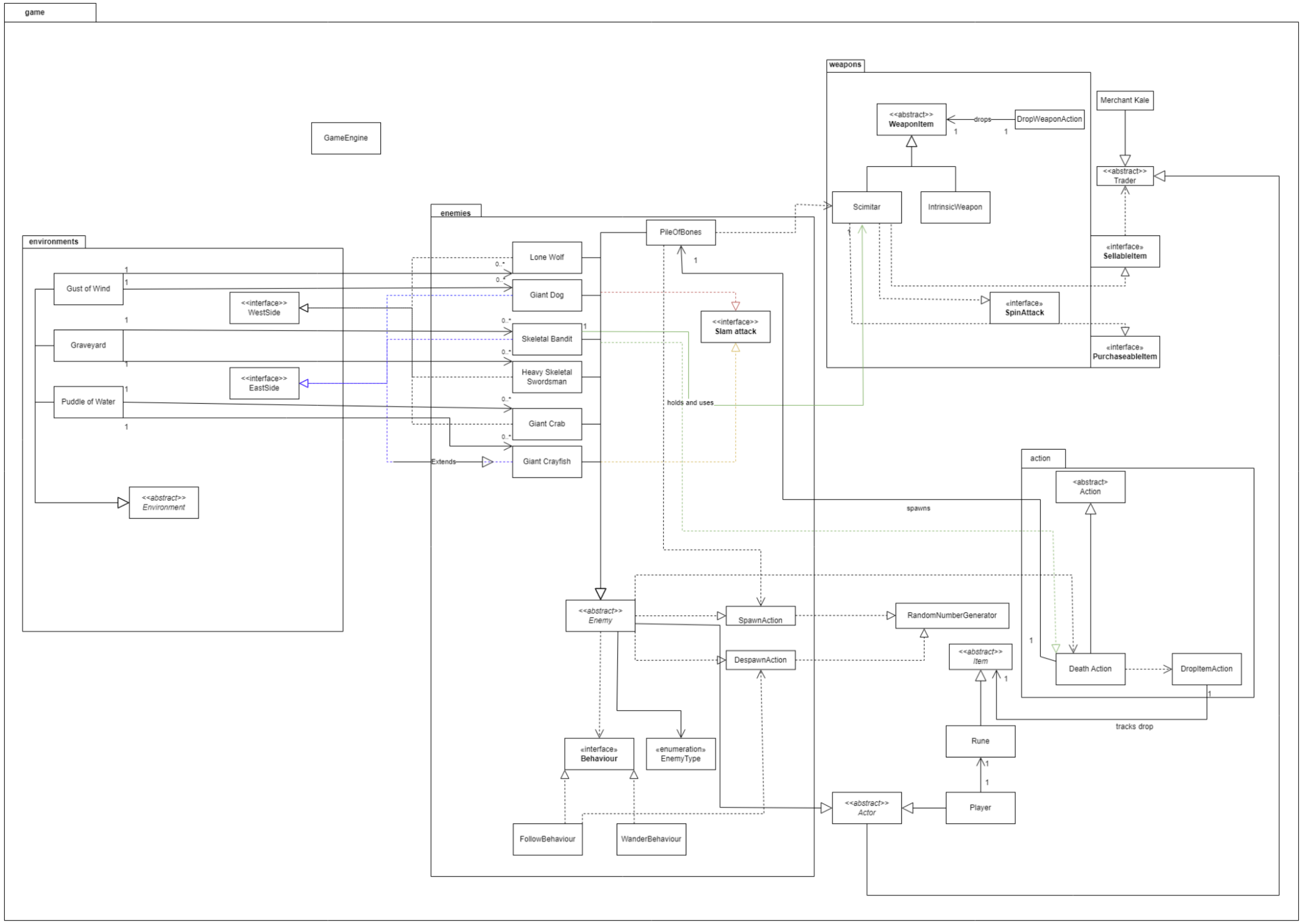
UML Diagram Requirement 3



UML Diagram Requirement 4



UML Diagram Requirement 5



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 environment 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 it's instantiation is never required. These child classes are inhabited by specific enemies.

The enemy classes; Giant Crab, Lone Wolf and Heavy Skeletal Swordsman, are extended front 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 extends 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 witht he Enemies class, allowing for enemies to be spawned during the game loop. There is also a dependency between the SpawnAction class and the RandomNumberGenerator class, this allows for instantiation of enemies at random times.

The child classes that extend the Enemies class includes 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.

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 actor classes Player, Enemies and Trader all extend the abstract Actor class as they are all entities that are alive in the system and require an inventory that stores items. As such, the DRY principle is upheld since they share common attributes and methods. In addition, items such as runes and actions such as the death action do not have multiple associations with every different actor class.

The abstract Actor class has an association with the Runes class. This enables all actor classes, Player, Enemies and Trader to interact with the currency, whether they drop it, pick it up or use it for trading purposes.

The AttackAction class has dependencies on both the Player and DeathAction class. This ensures 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 DespawnAction class has an association with the abstract Actor class and a dependency on the Runes class. In this way, the runes dropped by enemies that have despawned will disappear.

The Runes class also extends the abstract Item class. This ensures that it is an item, something that can rest on the ground, be picked up, dropped or be carried around in an actor’s inventory. The Runes class also has a dependency on the Random Number Generator class in order to account for the range of runes that an enemy may drop.

The Merchant Kale class extends the abstract Trader class. This ensures the avoidance of repetition for any future potential traders that we may add as they all would share the same attributes and methods as well as stay in line with the DRY principles.

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.

Requirement 3

The UML diagram demonstrates how the system will design the game reset and when the player dies using inbuilt engine classes to ensure an object-orientated functionality.

The Site of Grace, Player, GroundItems, DeathAction, and Location classes all implement the Resettable interface. This ensures that when the game is reset or the player dies, the player will respawn in the Site of Grace and all items on the ground disappear. This also upholds the Interface Segregation Principle, since it supports the use of small interfaces with one use. However, this neglects the Dependency Inversion Principle, as the classes are too tightly coupled with the Resettable Interface, making it difficult to change the implementation of the interface without affecting the other classes.

The Flask of Crimson Tear class extends the abstract ConsumableItem class. This allows the player to consume the item for its benefits and ensures that any additional consumable items may be included in the system through polymorphism. This also upholds the Open Closed Principle, as it ensures that no modifications are required when a new consumable item is added. However, it may disobey the Single Responsibility Principle, if more consumable items are added in the future that are unique to the rest. This may result in the abstract ConsumableItem class becoming a god class with multiple responsibilities.

The Floor class extends the abstract Ground class with a dependency to the Runes class as well. This will enable runes to be present on the floor of the game when they are dropped. This upholds the Open Closed Principle as it prevents multiple modifications to the abstract Ground class but still promotes the extension of more classes. However, this violates the Liskov Substitution Principle as Floor is a subclass of Ground, which may cause complexity issues in the future, as the child class cannot substitute for the Ground class, or parent class.

The Location class has a dependency on the abstract Item class to ensure that when a player dies, their runes will be left on the ground of a certain location.

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. Furthermore, the Grossmesser implements the SpinAttack interface, the Uchigatana implements the Unsheate interface, and Great Knife implements the Quickstep interface. As these are unique weapon skills they are specific to certain weapons, and such interfaces were the best choice for implementation.

The Quickstep interface has a dependency with the Location class, as Actor position is required to perform methods.

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.

The two sides of the map are implemented by creating two interfaces called WestSide and EastSide, which are then implemented by enemies that spawn on their respective sides. For example, a lone wolf, a heavy skeletal swordsman and a giant crab implement the interface WestSide as this will allow them to spawn on the west side of the map. Likewise, a giant dog, skeletal bandit and giant crayfish implement EastSide, enabling them to only spawn on the east side of the map. This is an example of the interface segregation principle because we are creating separate interfaces for east and west so that each enemy does not have to implement a method that says whether it can spawn in the west or east using location. A benefit of this implementation is that this ensures that interfaces stay small meaning that it is easier to add new features to the game without the need to refactor methods or attributes, hence making it extensible. For example, if a new enemy is to be added that can spawn on both sides of the map, we can simply ensure that the enemy implements both interfaces, which is simple. A con to this execution is that

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. Else, 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 can simply implement it for that new weapon, which benefits hugely in terms of simplicity.