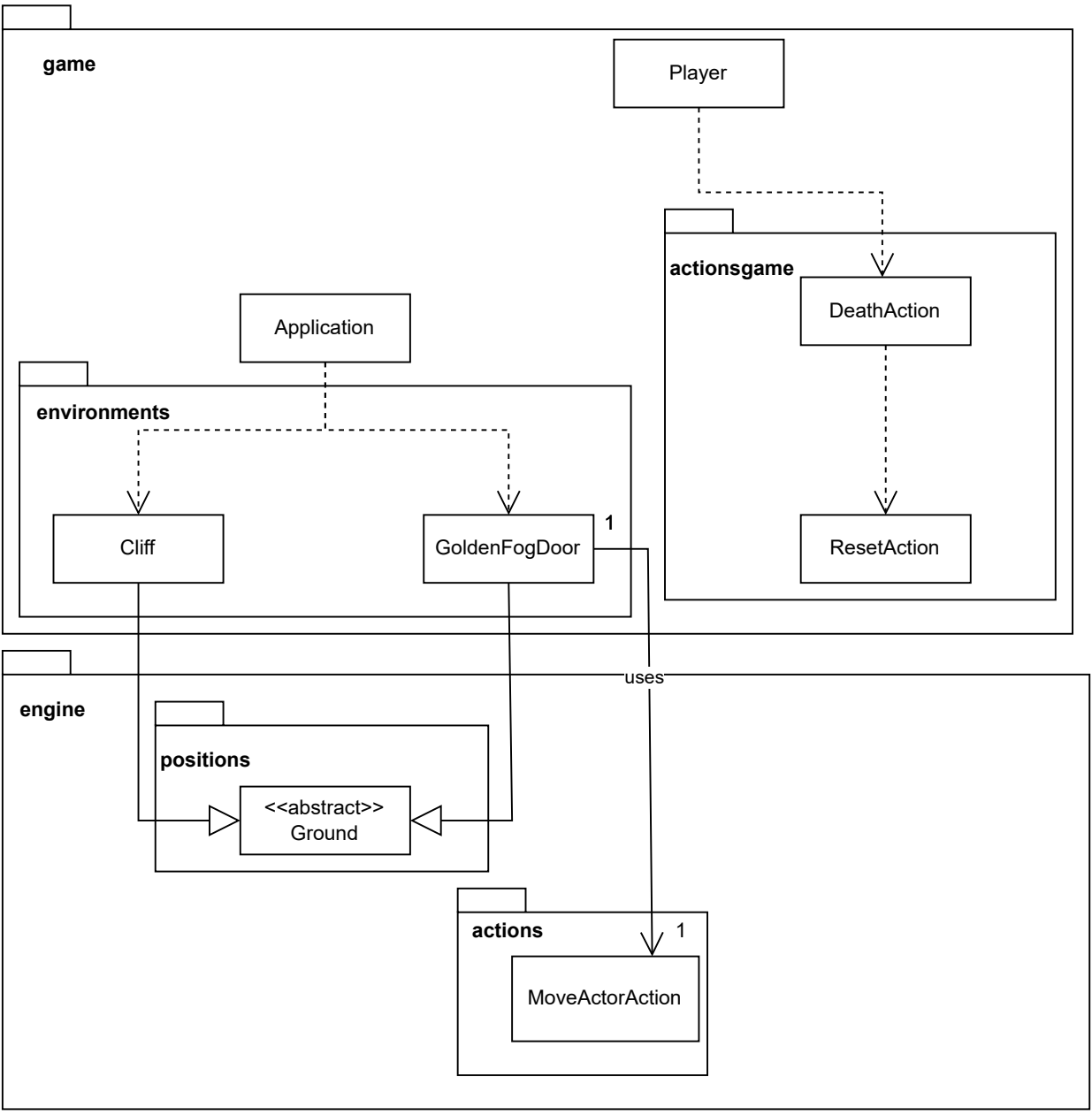
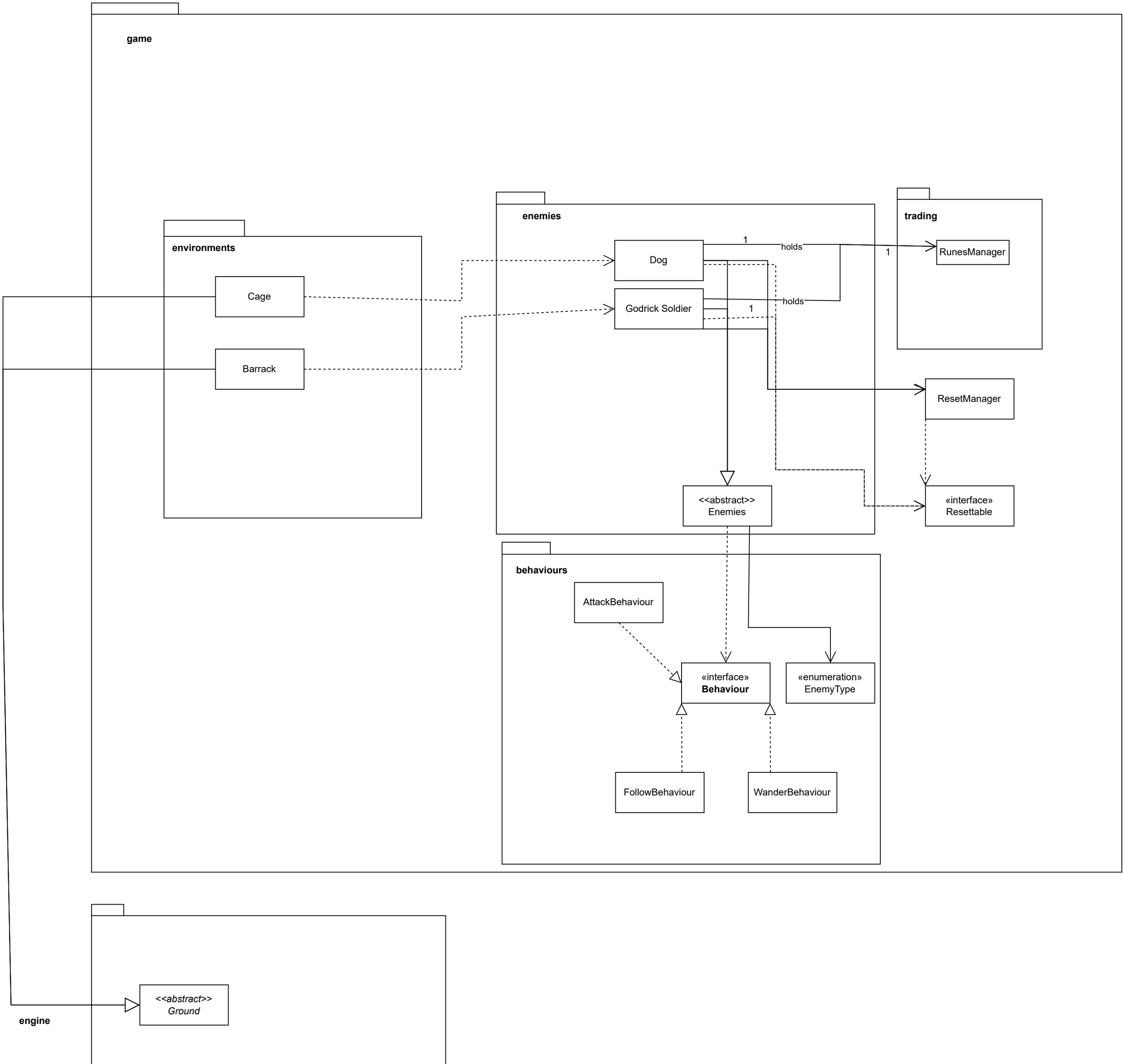
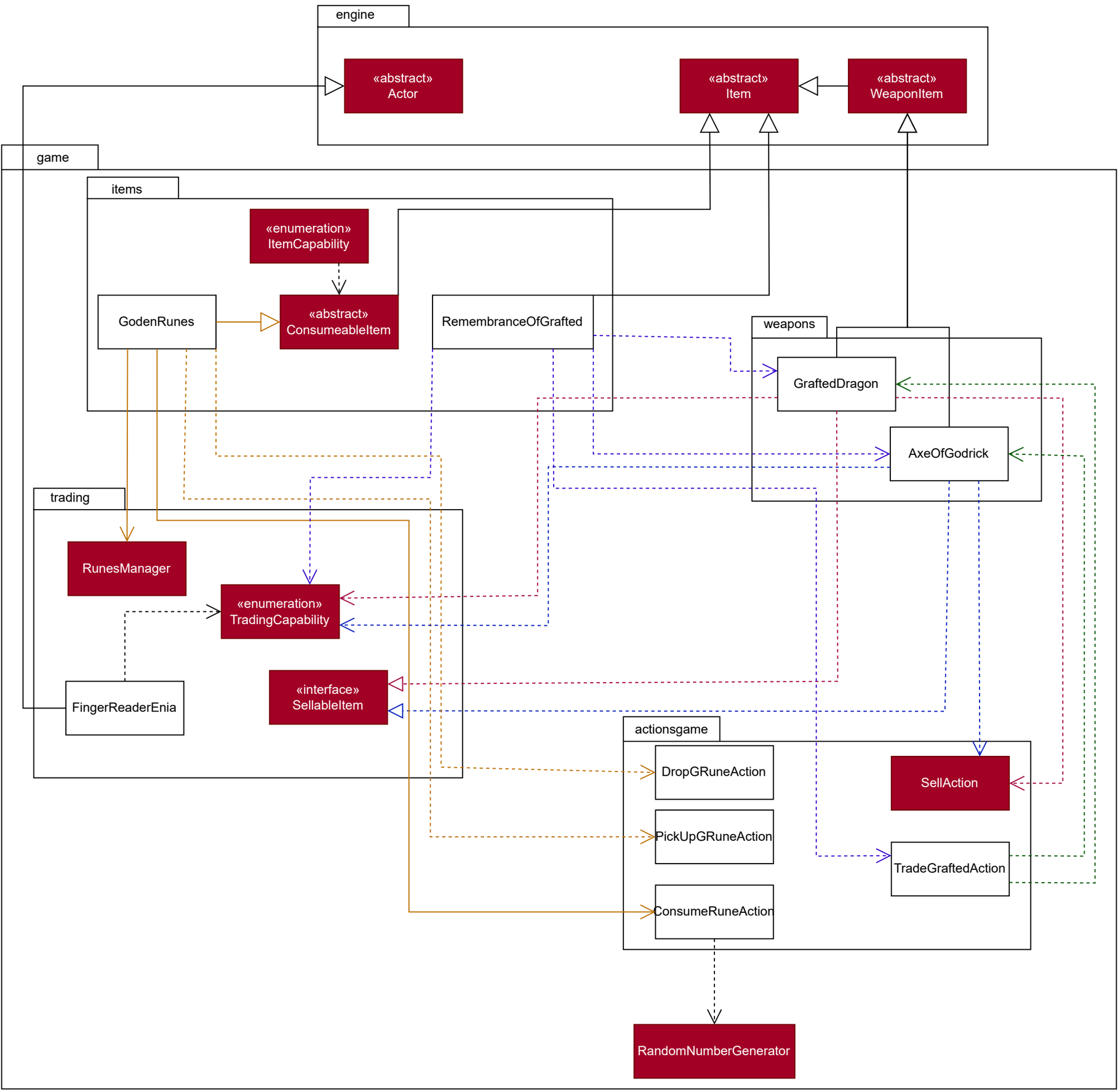
UML Requirement 1



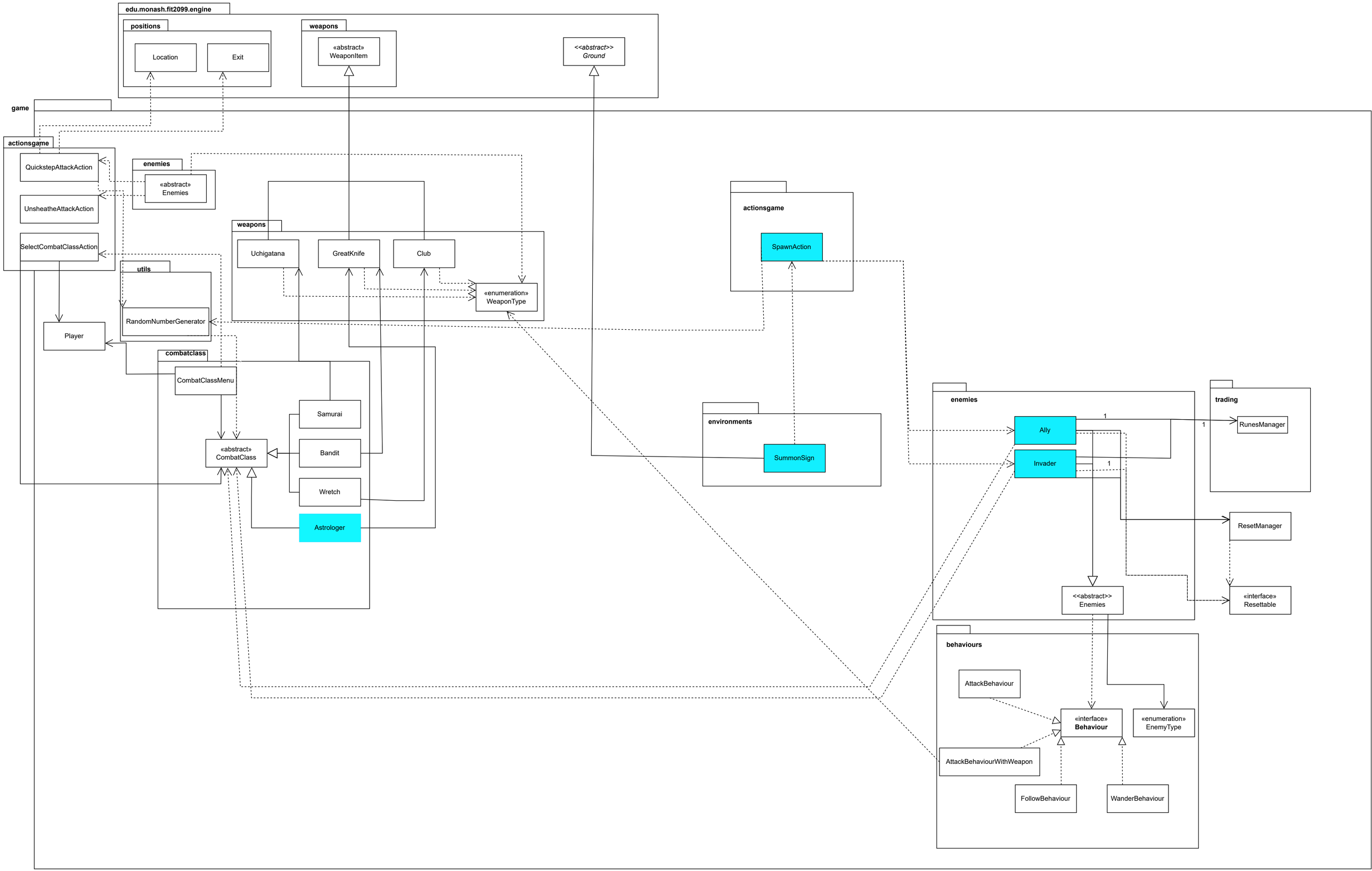
UML Requirement 2



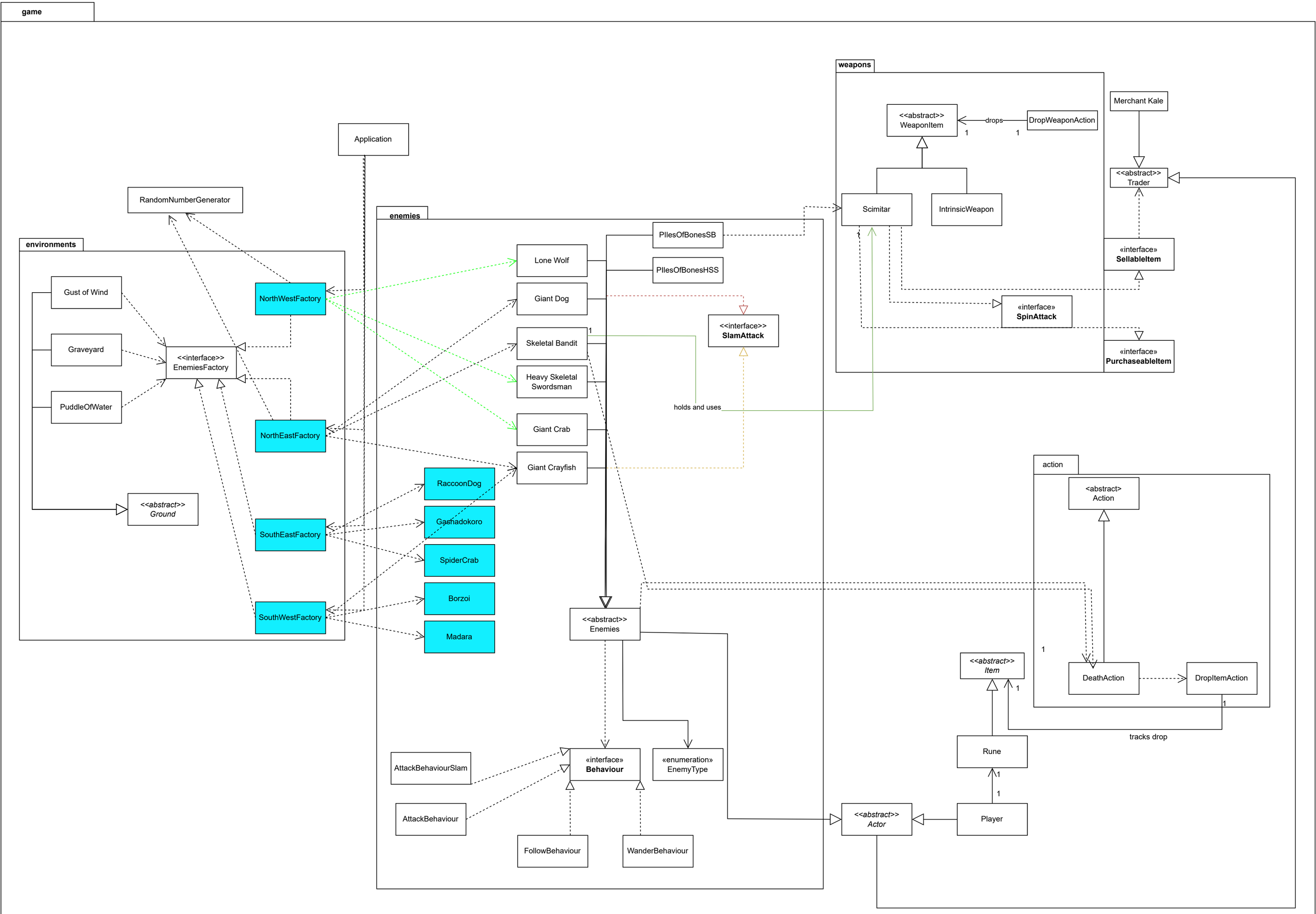
UML Requirement 3



UML Requirement 4



UML Requirement 5



A3 Requirement 1

The diagram demonstrates how the addition of new environments such as the Cliff and the GoldenFogDoor will interact with existing classes.

Both the Cliff and GoldenFogDoor extend the abstract class Ground since both classes have entities that appear on the map naturally when the game starts. By doing so, we uphold the Dependency Inversion Principle, since we enable additional ground-type classes in the future to be easily implemented. This also upholds the Liskov Substitution Principle as it ensures that any classes similar to these can inherit the same abstract class to acquire the same methods. However, it may breach the Single Responsibility Principle since all these environments require Ground to appear on the map, making Ground somewhat of a God class.

The association between GoldenFogDoor and MoveActorAction class enables the player to be transported from the door of the map they are currently on to the other door located on the new map.

The player class has a dependency on DeathAction class, which has a dependency on the Reset Action Class. This ensures that when the player becomes unconscious, after falling off the cliff, the death sequence and resets to the game will occur.

Pros
Addition of new ground classes that hurt or kill can be easily added in the future

Easily add more doors in the future to new maps and existing maps wherever needed

Cons
Since the death action occurs as a result of the player interacting with the ground, rather than the standard death of a player by another enemy killing them, it uses a different pathway to the death action and reset for the game

A3 Requirement 2

The UML diagram represents the additional enemy classes Dog and GodrickSoldier and the respective environments that they spawn in, Cage and Barrack to the existing system.

Both the Dog and GodrickSoldier classes extend the abstract Enemies class. In doing so, we uphold the Liskov Substitution Principle, this enables us to easily add even more enemies in the future. Alternatively, since both these enemies are enemies that only spawn in the Stormveil Castle map, another abstract enemy class could be made that specifically caters towards Stormveil Castle enemies.

Likewise, both the Cage and Barrack classes extend the abstract Ground class. This upholds the DRY principles. In addition, the Dog and GodrickSoldier classes both implement the Resettable interface. This ensures that the Interface Segregation Principle is upheld, as it utilises the same interface in order to reset and remove the enemies from the map when the game resets.

As the new environments are not separated into different areas, both the Cage and Barrack classes have a direct dependency on their respective Dog and GodrickSoldier classes. This ensures that each environment only spawns that specific enemy when it appears on the map. This prevents the breach of the Open Closed Principle since we do not use any if-else statements to handle which enemies spawn where.

Pros
Allows more enemies and their respective environments to be added in the future

Cons
New enemies only spawn in a specific map and nowhere else, so we could instead make a new abstract enemy class specifically for them

A3 Requirement 3

The UML diagram sees the introduction of the new trader Finger Reader Enia, and items such as Axe of Godrick, Grafted Dragon, Golden Runes, Remembrance Of Grafted and their associated actions.

The Axe of Godrick and Grafted Dragon are weapons that extend the WeaponItem parent class, upholding LSP, meaning that any methods that require a WeaponItem class can accept these two as parameters too. These two weapons further implement the sellable interface. This ensures that it aligns with the current game system, allowing for the weapons to be sold, and further upholding the open/closed principle by allowing for the classes to be extended within the existing system. Furthermore, it was decided that both these classes would have a dependency on the SellAction class, to follow the selling method implemented by previous weapons. There is a further dependency from the TradeGraftedAction on the weapons, ensuring that the weapons can only be acquired through that action. There is also a dependency from these weapons to the TradingCapability enumeration, this is done to follow the existing system and the method of trading previously set up.

The Remembrance of Grafted extends the abstract Item class, once again upholding LSP and the open/closed principles. This allows for the item to be used throughout the existing system, and further allows for extensions to the normal Item class behaviours, such as trading for Godrick's weapons. There is a dependency from the Remembrance of Grafted to Godrick's weapons, allowing for the instances to be used in the relevant trade. There is a further dependency on the TradeGraftedAction, further following the implementation given by the existing system.

Golden Runes extends the existing Consumable Item, an abstract class, also being an extension of the Item class by inheritance. This allows for all the functionality of an Item and Consumable Item, with the added benefit of having extended concrete functionality. This upholds the open/closed principle and further upholds LSP as this can be used in any method that requires the Item or Consumable Item class. There are dependencies to the DropGRuneAction and PickUpGRuneAction classes, once again this is done to align with the existing system. There is however an association with the ConsumeRuneAction, this is done to ensure that the referenced action is only associated with a given instance of a Golden Rune. Furthermore, there is an association with the RuneManager, this is to ensure that the transfer of runes is solely done by it, upholding the single responsibility principle.

Finger Reader Enia class is another trader, that extends the Actor abstract class, once again upholding the open/closed principle. Its only dependency is on the Trading Capability enumeration, where the class' trading capability is defined. Due to this rather than have all the sell and trade functionality in rooted in the class, it is outsourced to the various actor classes, to allow for trading actions.

Pros:
This can allow for further unique trades with Finger Reader Enia or other traders given the correct capability. Also allows for further extension of Consumable Items that can improve different player stats, not just limited to runes and health.

Cons:
Finger Reader Enia has similar functionality to Merchant Kale, their functionalities can be further abstracted into a trader class. This will make the creation of new traders much easier, rather than relying on an extension of the Actor abstract class.

A3 Requirement 4

In this requirement, it builds onto the new combat archetype adding the class Astrologer, and also a new environment that allows the player to add a randomly placed Ally or Invader.

Like the previous implementation of classes, it was easy to extend to this new class, because all it does is pass in HP, class weapon, class name and display the character of Astrologer. It should be noted that we have opted not to make the Astrologer's Staff and have given the class, a GreatKnife. As such it is an example of the DRY principle as we are not repeating ourselves. Furthermore, if another class was to be added, all we would have to do for Invader and Ally to access the new class is to add it to the getRandomCombatClass in RandomNumberGenerator.

In the implementation of the SummonSign environment, I choose to do the summoning of Allies and Invaders through an action. As this environment didn't require different forms of enemies spawning in each area, an enemies factory was not needed, furthermore, if the environment was to be extended to spawn more different entities, we can simply add it to the action, making it OCP such that it is extendable.

Lastly, the implementation of Ally and Invader extends from the Enemies class, and for HP we would get the CombatClass' max HP and for the weapon, we would add their respective weapon from CombatClass to the inventory. As such upholds the DRY principle as stated before in requirement 1 of assignment 2.

Pros:
If a new type of entity was to be added to SummonSign it would be very easy to implement, as we extend from the Enemies class, and implement in the action.
Cons:

If we wanted SummonSign to summon different entities in different areas of the map, we would need to create an enemies factory and this would require a lot of work, as well as reworking the action.

A3 Requirement 5

In our creative requirement, we decided to divide the map into North-East, North-West, South-East and South-West instead of just West and East. Also to account for the new areas, we created 5 new enemies of the name, RaccoonDog, Borzoi, Madara, SpiderCrab and Gashadokoro.

Due to our earlier efficient and clean implementation of requirement 5 in assignment 2, it was very easy to extend the two regions to four regions. This would require creating two new enemies factory namely, SouthEastFactory and SouthWestFactory. We renamed East to NorthEastFactory and West to NorthWestFactory. So, each of these factories would implement the EnemiesFactory, and depending on the given environment if given a certain direction factory would summon that corresponding enemy in that environment. Thus, this upholds the interface segregation principle as if we were to create new environments to spawn a new type of enemy, we could simply implement a new enemies factory and that would enable that.

Implementing the new enemies was easy as all of them extend of the enemies class, and as stated before upheld the DRY principle.

Pros:
Easy to implement due to the use of enemies factory which allows us to extend the regions.

Cons:
Need to create a new directional factory each time a region is created.