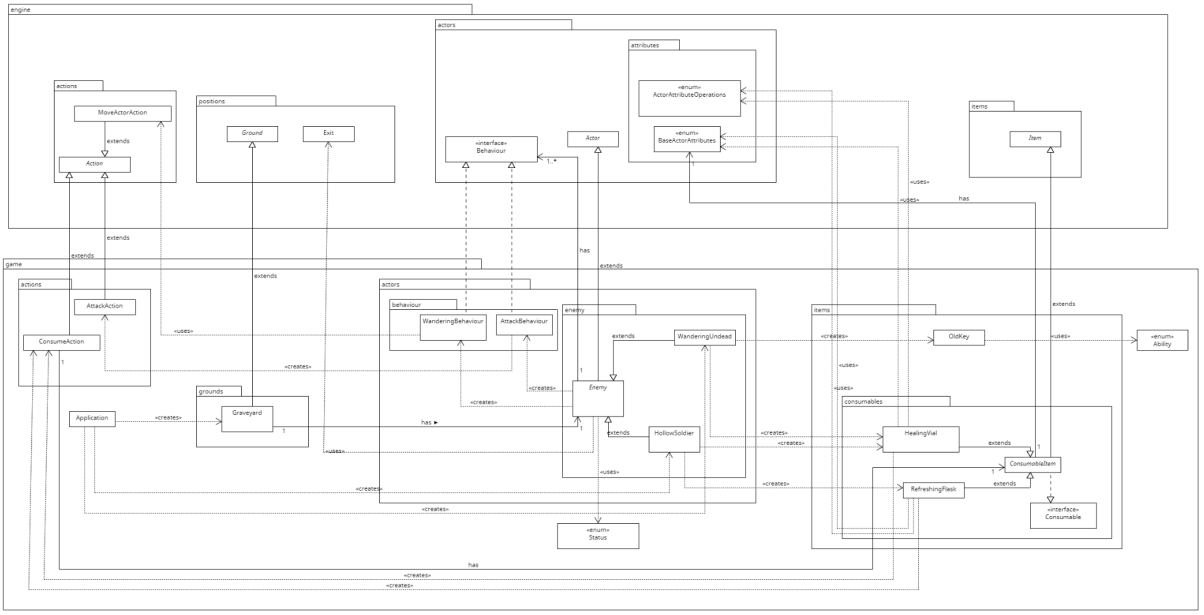
REQ5 UML Diagram.



The primary objective of REQ5 is to introduce a new enemy called Hollow Soldier. Hollow Soldiers can drop healing vials and refreshing flasks upon death, and can be spawned by graveyards. Healing vials may be dropped by Wandering Undeads as well.

To accommodate the new gameplay elements, I made a HollowSoldier Class, which extends the abstract Enemy class, much like the WanderingUndead class. The abstract Enemy class utilises the Status enum class to determine which actors it should target for attacks, adhering to the Open/Closed Principle (OCP). The alternative would have been to use if-else statements and instanceof checks, which would have made the code less maintainable and extensible.

Since our implementation of Graveyard adheres to the Dependency Inversion Principle (DIP), Graveyard relies on the abstraction of an ‘Enemy’, rather than its concrete implementation to spawn enemies. As such, we do not have to modify any of our Graveyard class code to accommodate this new enemy. (OCP)

I created a ‘Consumable’ interface to implement a ‘consume’ method. This implementation reduces code duplication (DRY) and simplifies future extensions, as future consumable items can just implement this interface.

I also created an abstract ConsumableItem class, which extends the abstract Item class and implements the Consumable interface. This setup ensures that new ConsumableItem subclasses only need to implement relevant interface methods, respecting the Liskov Substitution Principle (LSP).

RefreshingFlask and HealingVials extend the ConsumableItem abstract class. This design reflects their shared characteristics as consumable items with restoring abilities.

To enable item consumption by the player, I made the ConsumeAction class, extending the abstract Action class. This class has an association with the abstract ConsumableItem class, this allows different ConsumableItem subclasses to execute their own implementation of the ‘consume’ method inside of the ‘execute’ of ConsumeAction. Additionally, ConsumableItem and its subclasses depend on the BaseActorAttributes enum and ActorAttributeOperations enum to update the player's attributes after consuming items with restoring abilities.

Both RefreshingFlask and HealingVials classes establish a dependency relationship with ConsumeAction, enabling actors who possess these items to consume them.

Advantages:

Reduced Code Duplication

The use of the Consumable interface and abstract ConsumableItem class for consumable items helps reduce code duplication by providing a common structure and methods for all such items. (DRY)

Simplified Future Extensions

By using interfaces and abstract classes, this implementation creates a flexible framework for adding new enemies (Hollow Soldier) and consumable items (Healing Vials, Refreshing Flasks). This makes it easier to extend the game's functionality without major modifications to existing code, adhering to the Open/Closed Principle (OCP).

Disadvantages:

Increased Number of Classes

Introducing abstract classes and interfaces for every entity type like Enemy, ConsumableItem, etc. can result in a large number of classes in the codebase. Which may be hard to navigate and maintain.

Maintenance

Changes to the core abstractions, interfaces and abstract classes, can have ripple effects throughout the codebase. Which may lead to harder maintenance when modifying these abstractions down the line.