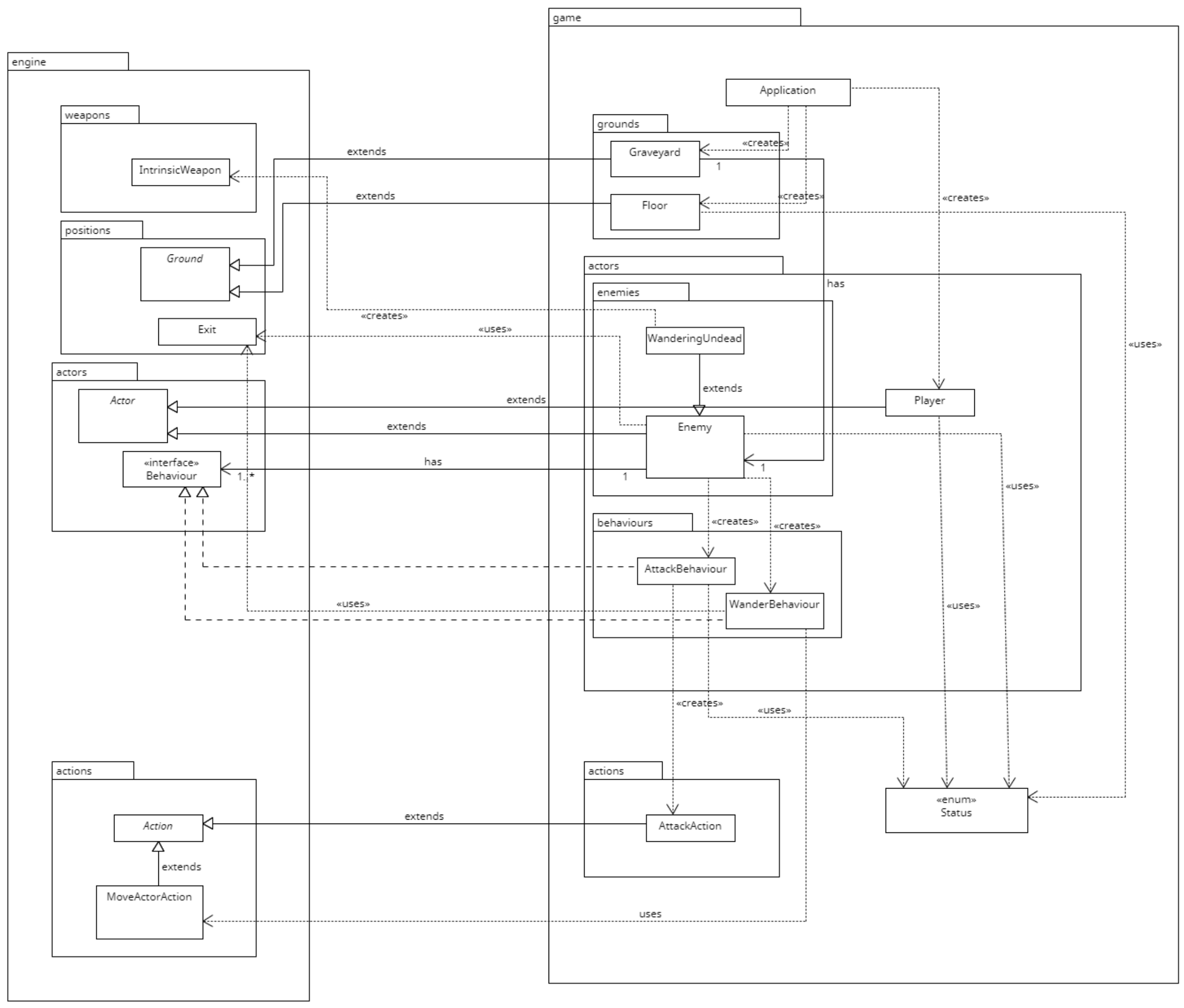
REQ3 UML Diagram.



The primary objectives of REQ3 are to define various behaviours for the Wandering Undead, such as attack and wander, and to only allow Player actors to enter ‘floor’.

To achieve this, I made Floor class extend the abstract Ground class and depend on the Status enum class to determine if the actor is a Player, only then will the actor be allowed to enter. This implementation follows OCP (Open-closed Principle) since we can add more enemies into the game and not have to modify the existing code for Floor. The alternative to this would be to use if-else statements and the ‘instanceof’ operator to check if the actor in question is a Player, which would increase dependencies.

Since enemies share common behaviours, I made WanderingUndead class extend the abstract Enemy class. Since they share some common attributes and methods, abstracting these identities avoids repetition, adhering to the (DRY). Abstract Enemy class also has a dependency on the Exit class to assess the presence of other actors in its vicinity, this determines whether it attacks or just wanders. It depends on the Status enum class to ensure it only attacks Players, not other Enemies.

The abstract Enemy class has an association relationship with the Behaviour interface, which is implemented by the AttackBehaviour and WanderBehaviour classes.

The AttackBehaviour class will handle attack actions that are executed by the enemies and not the Player. It has a dependency relationship with AttackAction as it creates an AttackAction instance to perform the attack. WanderBehaviour class handles the wandering movement of all enemies, it has a dependency relationship with the MoveActorAction.

This implementation follows OCP since we don't have to modify the existing code to accommodate new behaviours added in the future. It also aligns with the Dependency Inversion Principle(DIP) where all behaviours depend on the abstraction of 'Enemy' instead of its concrete implementations.

Advantages:
Allows for easy extension of new enemies

Since this implementation does not rely on concrete instances but rather abstractions of Enemy, the logic of only Players being allowed to enter floors will be preserved without modification when new enemies are added.

Code reusability

With an Enemy abstract class, we can encapsulate common behaviour and attributes of Enemies, which means we avoid repeating ourselves (DRY), when adding new enemies down the line.

Disadvantages:
Limited Flexibility for Unique Enemy Behaviours

While the implementation allows for the easy addition of new enemies with common behaviours, it might become challenging if we need to introduce enemies with highly unique or specialised behaviours. In such cases, we may be constrained by the shared abstractions in the Enemy class and may need to make significant modifications.

Increased Complexity

As the number of behaviours and enemy types increases, the codebase could become more complex and harder to maintain. Managing a large number of behaviour classes and ensuring that they work correctly in different combinations might lead to increased complexity and potential for bugs.