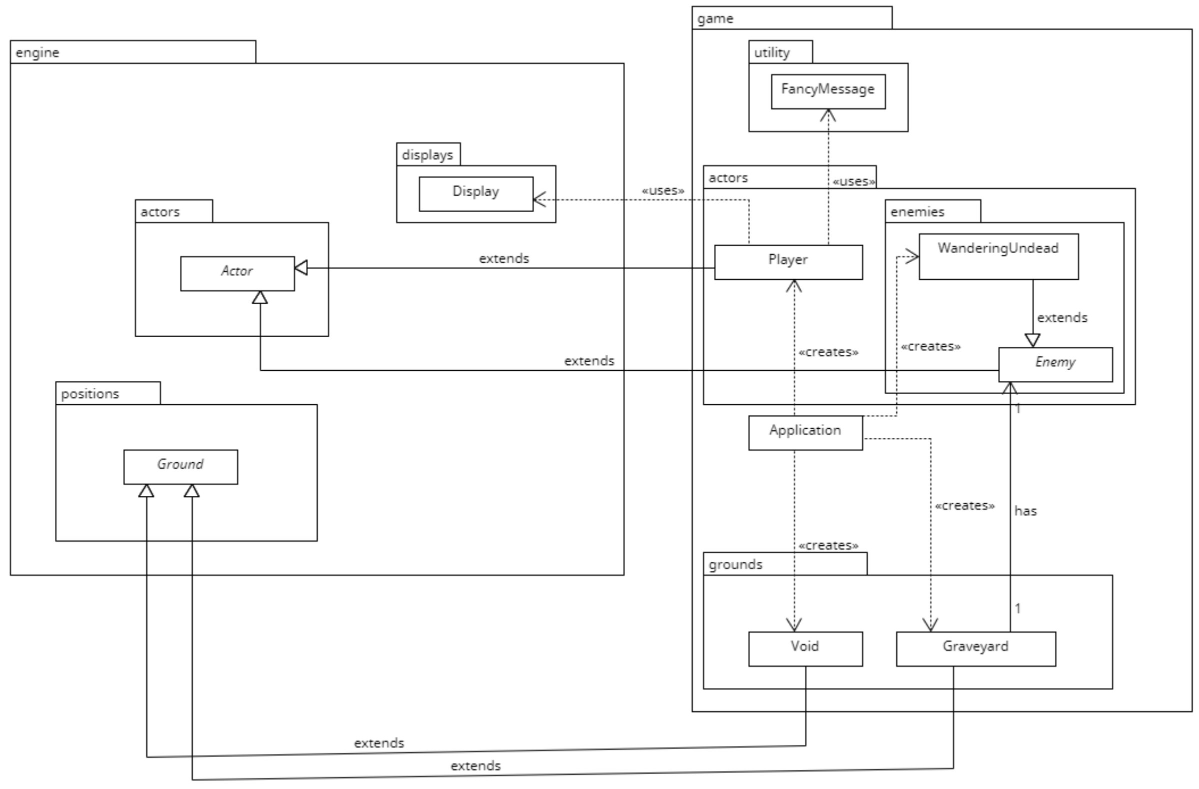
REQ2 UML Diagram.



The primary objectives of REQ2 are to implement the Void that kills actors if stepped on, and introduce Graveyard that has a chance to spawn an enemy called WanderingUndead.

To accomplish this, I made the Void class and Graveyard class extend the abstract Ground class, as they share common attributes and methods, we need not repeat the code(DRY). Both classes can override these methods to implement different functionalities. In this implementation, the Void class overrides the tick method to check if an actor steps on it, if so, it makes the actor unconscious. The Graveyard class overrides the tick method to have a chance to spawn enemies during every turn of the game. This implementation follows the Open-closed Principle (OCP) since Void class and Graveyard class extend the functionality of the ground without needing to modify existing code.

I made all enemies extend the abstract Enemy class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY).

Graveyard class has an association relationship with the abstract Enemy class, it accepts an Enemy as an argument to spawn the Enemy, this adheres to the Dependency Inversion Principle (DIP) as Graveyard relies on the abstraction of an 'Enemy', rather than its concrete implementation to spawn enemies.

When adding a new enemy that’s spawnable by graveyard to the game, we only need to make that new enemy class extend the existing abstract Enemy class, we do not need to modify the existing Graveyard class, this adheres to (OCP).

The implementation aligns with the Liskov Substitution Principle (LSP) since we can substitute any Ground object with a Void or Graveyard without causing unexpected issues.

This implementation does not rely on techniques like the instanceof operator or checking the class name (object.getClass().getName()) to determine the specific subclass of enemies before spawning them. This practice ensures that derived classes can seamlessly substitute their base class counterparts, preserving the expected behaviour and adhering to the principles of LSP.

Alternative implementation:

Use an interface called ‘Spawner’ that provides a contract that handles the spawning logic of enemies, graveyard and future elements can implement this interface and its methods. This alternative may provide more flexibility and decoupling, when adding different elements that spawn enemies in the future. However, it may also increase complexity. Depending on the complexity of spawning logic, implementing the 'Spawner' interface in various classes could make the codebase more complex and harder to maintain. Each implementing class would need to handle spawning differently, which might lead to confusion.

Advantages:

Code reusability

With an abstract class, we can encapsulate common behaviour and attributes within the base class, promoting code reusability among subclasses.

Maintenance and Updates

When gameplay mechanics or enemy types need to change or be updated, we only need to modify the abstract Enemy class, as it propagates these changes to all enemies. This reduces the risk of introducing bugs and simplifies maintenance if we add many more enemies down the line.

Disadvantages:

Inheritance hierarchy limitations

Java only allows for single inheritance, which means that if enemies need to inherit from another class other purposes unrelated to the common attributes and methods shared among enemies, using an abstract class as a base for enemies can limit class hierarchy. This restriction can make it challenging to incorporate additional features or behaviours that would require a different base class.

Restricted extensibility:

If we wanted to add enemy types that may have unique behaviours that don't fit neatly into the common abstraction, this implementation can make it challenging to reuse the abstract class for diverse enemy types, potentially leading to a proliferation of specialised subclasses.