Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 2: The Abandoned Village's Surroundings

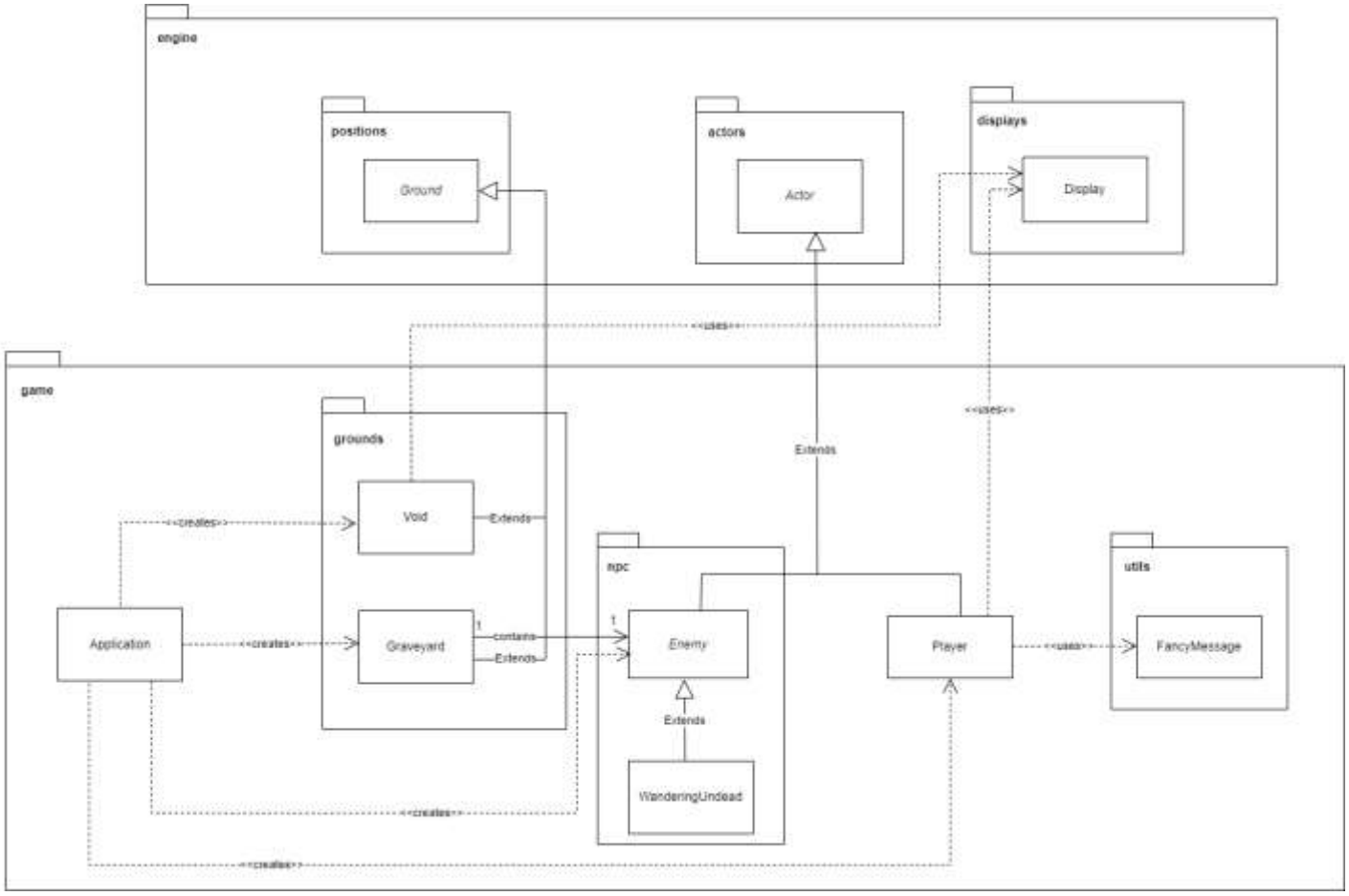

Figure 1: UML of REQ2 - The Abandoned Village's Surroundings

The provided diagram depicts the entire system implementation for Requirement 2, which involves the Abandoned Village's Surroundings. The primary objective of this design is to introduce new classes, specifically the Void class to eliminate all entities that step on it and the Graveyard class to spawn enemies during each turn of the game. In this extended system, three new concrete classes (Void class, Graveyard class, and WanderingUndead class) have been introduced, along with one abstract class called Enemy.

We decided to have both the Void class and Graveyard class extend the abstract class Ground from the engine to avoid redundancy and aligns with the Don't Repeat Yourself (DRY) principle. This decision stems from the fact that these classes share common attributes and methods. Additionally, the Graveyard class has an association relationship with the abstract Enemy class to spawn different enemies at certain chances during each turn of the game. This design choice reduces multiple dependencies (ReD) on various classes that may be implemented in the future, such as the WanderingUndead class. Importantly, this approach adheres to the Liskov Substitution Principle (LSP) since it doesn't violate LSP by using the instanceof operator or object.getClass().getName() to identify the actual subclass before spawning enemies.

The advantage of treating both Void class and Graveyard class as types of ground is that neither of them needs to move, and both can access relevant methods from the abstract Ground class. They can override these methods to implement different implementations. For example, the Void class can access the tick method to track whether an actor steps on it and immediately eliminate the actor, while the Graveyard class can implement the tick method to spawn enemies during every turn of the game, regardless of the player's location. This design aligns with the Open/Closed Principle (OCP) since Void class and Graveyard class extend the functionality of the ground without modifying existing code. However, a potential disadvantage is that code maintenance might become challenging as the system grows larger and more complex.

Furthermore, we have designed the Enemy classes as an abstract class that extends the abstract class Actor from the engine. All inhabitants or enemies in the game will extend this Enemy abstract class because they share common attributes and methods. Enemies in the current implementation, Wandering Undead will extend this Enemy abstract class. This approach ensures that when additional requirements are introduced in future game implementations that apply to all enemies, these new features can be added to the abstract Enemy class, preventing code repetition (DRY). Nevertheless, a drawback is that all subclasses extending from the Enemy abstract class must implement any abstract methods defined in the class, potentially leading to unnecessary implementations in those classes.

Additionally, the Void class uses the Display class to show a message indicating the actor's death. The Player class has dependencies on both the Display class and FancyMessage class to print the death message in the console menu. The Application class creates Void class, Player class, Enemy class and Graveyard class, so the Application class has dependency with those classes.