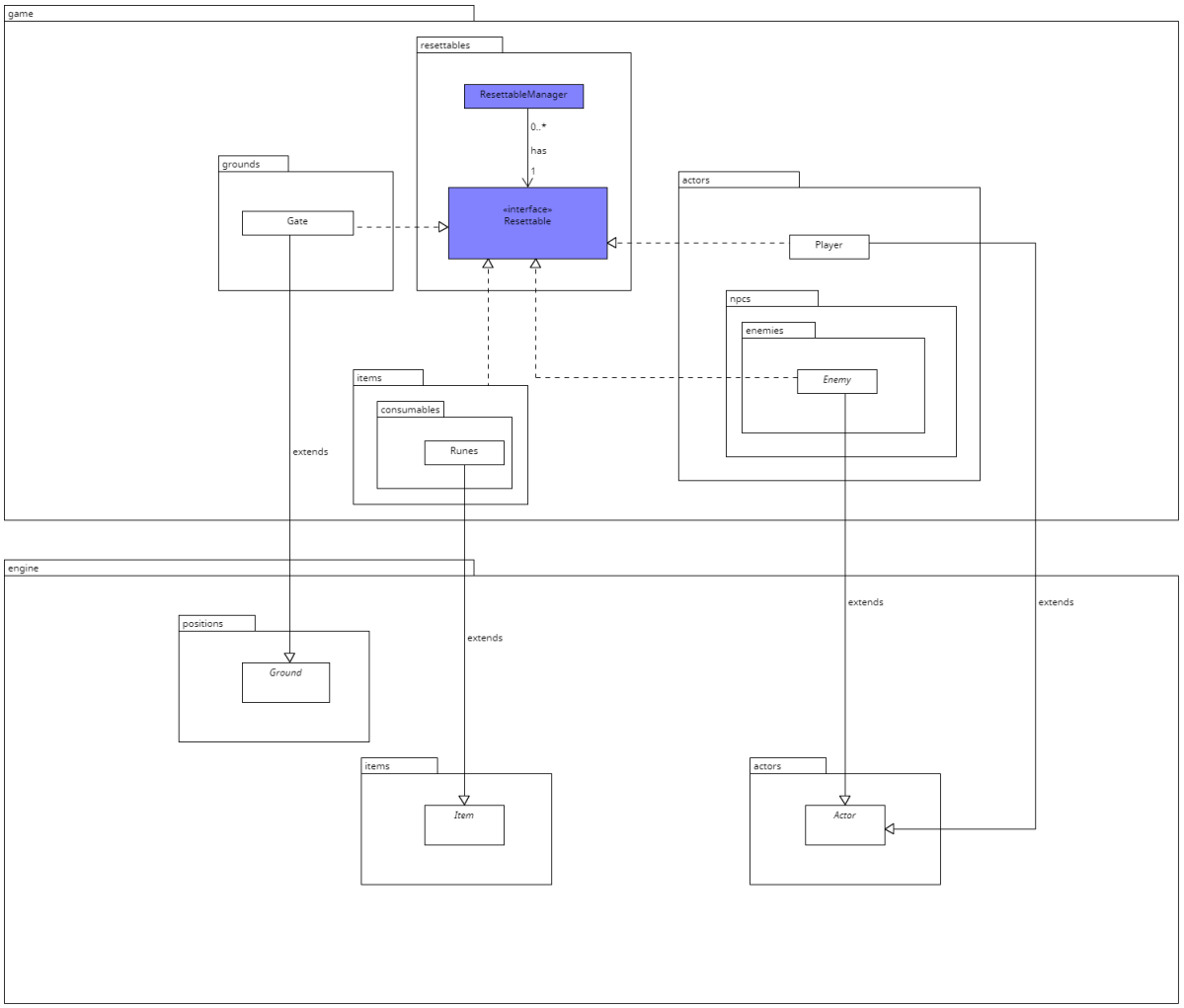

The main objective of this task is to implement the feature of resetting all entities on the map when the player dies.

To achieve this, we created a ‘Resettables’ interface, all entities that can be reset: Runes, Gate, Enemy, and Player, will implement this interface. This interface has a method called reset(), that is implemented by these classes. This adheres to the Interface Segregation Principle (ISP) as we created a specific ‘Resettable’ interface that contains only the method relevant to the implementing classes, and we do not force entities that do not need to be reset to implement this method.

We created a ‘ResettableManager’ class that has an array list of ‘Resettables’, which represent the resettables that are registered to the manager. This adheres to the Dependency Inversion Principle (DIP) as ‘ResettableManager’ relies on the abstraction of ‘Resettables’ rather than its specific implementations. ‘ResettableManager’ also has static methods of ‘addResettable’ and ‘removeResettable’ that are used to register and remove resettables from the manager. Lastly, it has a ‘executeReset()’ method that loops through all the resettables that are registered to the manager (the resettables inside the array list), and calls their respective reset() methods. This adheres to the Single Responsibility Principle (SRP), as ‘ResettableManager’ is only responsible for managing the resettables in the game.

This implementation also adheres to the Open-Closed Principle (OCP) as we can add new resettable entities in the future by creating new classes that implement the ‘Resettables’ interface without changing the existing code.

Advantages

Ease of Extensibility

Our implementation allows easy extensibility for more entities that can be reset upon the player’s death in the future, all they have to do is implement the ‘Resettables’ method, and we do not have to change any of the existing code.

Reduced Coupling

The use of interfaces and abstractions in adherence to the Dependency Inversion Principle reduces the coupling between classes. This means that changes to one class have less impact on others, making our system and game more resilient to change.

Disadvantages

Performance Overhead

Using interfaces and abstractions can introduce a slight performance overhead due to dynamic method dispatch, which may degrade performance of our game in the long-run.

Complexity

Introducing interfaces, abstractions, and manager classes can sometimes introduce unnecessary complexity, especially in smaller projects, and this can make it harder for new collaborators to pick up on the codebase.