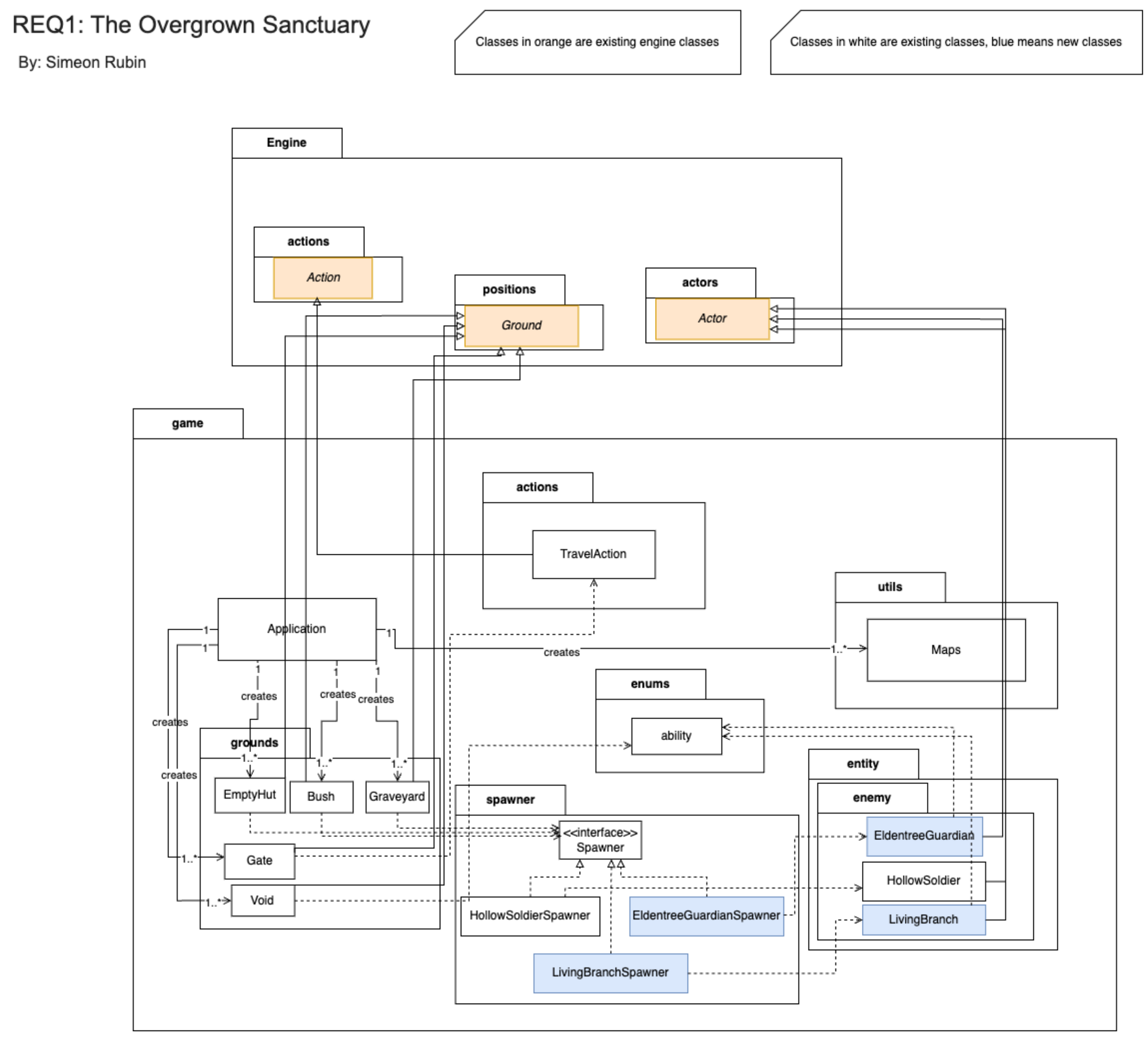
REQ1 Rationale



Overview

The UML design for the requirement reflects adjustments and a refactoring of the functionality of the Gate class to accommodate multiple destinations for the Player to travel to, as well as the addition of a few new enemy types and a new map.

There are 4 new classes introduced to meet this requirement which include 2 new enemy types and 2 new spawner classes.

Gate Refactor

In order to achieve the new multi-destination functionality for the Gate class, I considered a variety of possibilities for refactoring. This included the creation of a new DestinationHandler class which would take the GameMaps as input and then would be parsed into the Gate constructor and would output those destinations into a new TravelAction for each destination. The downside of this design is that it would involve the creation of a class to

handle functionality which could very simply and easily be handled inside the Gate class without violating the SRP, involving an overloaded constructor with an option of an ArrayList of GameMaps to be parsed into the second Gate constructor.
Since this second design option is simpler, doesn't involve having to redo previous implementations of the Gate class, doesn't involve unnecessary additional classes and still has the Gate class adhere to the SRP, it was chosen for my implementation.

The allowableActions() method of the Gate class will check if multiple destinations are being used and if they are, it will loop through the list of destinations and create a new TravelAction for each of them. Given the Gate and TravelAction classes' adherence to the OCP, extending the travel functionality without heavily modifying the core functionality keeps the new updates consistent with previous implementations of the travel feature.

Overgrown Sanctuary

The implementation of the new map simply involved the inclusion of the new map in the Maps utils class and the parsing of this map into the constructor of the Gate class to allow the Player to travel to this new location.

It is populated with different ground types which use a Spawner interface in the constructor to spawn a previous and some new enemy types.

New Enemy Types

The Eldentree Guardian and Living Branch enemy types both use standard enemy functionality with a new ability capability VOID_IMMUNITY added to them and to the Aberxyver class to add consistency to the functionality of immunity to the void (previously there was only an enum for the Aberxyver enemy type used to represent immunity to the void in the Void class).
They each have their own Spawner classes.
The use of a spawner class for each enemy, as opposed to instantiating them inside the relative ground types which spawn them (i.e. EmptyHut, Graveyard..) creates dependency injection where the use of these enemy types and their class functions is decoupled from their instantiations. This allows the design to follow the SRP, where the enemy type classes handle the behaviour and functionality of theses enemy types and the spawners are responsible for instantiation of them and DIP, where the lower level spawner classes depend on an abstraction (the Spawner interface).