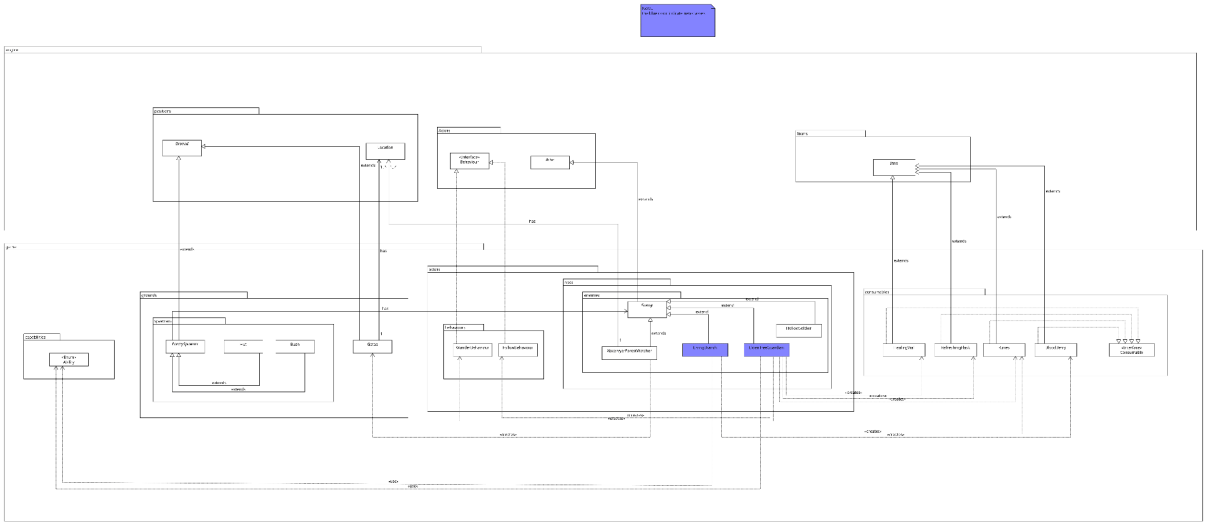

The main objective of this requirement is to add the feature of having multiple travel options for gates, and 2 new enemies, the Eldentree Guardian and Living Branch.

To achieve this, we decided to overload the Gate constructor. In addition to the original Gate constructor of accepting 1 Location and 1 string indicating the destination's name, we have another constructor that takes in a hashmap, with the Location as the key, and the destination's name as the value.

We decided on this implementation because constructor overloading can help achieve Single Responsibility Principle (SRP) by allowing us to create multiple constructors, each responsible for initializing an object with a specific set of properties or configuration. This separates the responsibilities of constructing an object from the responsibilities of the object itself.

Furthermore, by providing multiple constructors, we can avoid the need for a single constructor to take a long list of parameters or complex logic for conditional initialization. This makes the code more maintainable and easier to understand.

For the new enemies, we just create 2 classes: Eldentree Guardian and Living Branch, and just have them extend the existing Enemy abstract class, since they all share some common functionality. By doing this, we avoid code repetition (DRY).

Alternative implementation:
Initially, our idea was to modify the existing constructor and have it take in an arraylist of Gamemaps, an arraylist of Locations, and an arraylist of strings representing the names of the destinations. However, we quickly realised that this would be a big hassle when trying to add gates into the game as we would have to create 3 arraylists for each gate inside the Application class. As such, we decided against this implementation.

Alternative implementation #2:

Another idea we had was to refactor the existing constructor, making it take in a HashMap, with the Locations being the keys, and the strings representing the destinations being the values. However, this would be a potential violation of OCP, OCP states that classes should be open for extension but closed for modification. By refactoring the existing constructor, this would be a potential violation of OCP. This could lead to compatibility issues with existing code that relies on the previous constructor signature, and would require refactoring the gates that have been initialized in the previous assignments. As such, we decided against this implementation as well.

Advantages:

Version Compatibility:

By creating additional overloaded constructors while keeping the existing ones intact, we preserve backward compatibility and follow the Open/Closed Principle (OCP). Old gates that were initialised before this assignment will still work as expected, without needing to be refactored.

Avoiding Long Parameter Lists:

Without constructor overloading, we may need to end up with constructors that take a long list of parameters, which can make the code harder to read and maintain. Constructor overloading allows us to avoid this issue by providing multiple constructors, each taking a smaller, focused set of parameters.

Disadvantages:

Code Duplication:

Constructor overloading leads to a bit of code duplication as the initialization of attributes needs to be repeated in multiple constructors.