Design Rationale for Requirement 1: The Overgrown Sanctuary

New Gate Implementation

To ensure multiple destinations could be reached through one gate, an ArrayList of actions was added to each instance of a gate. Then, when a new Move Action was added to a gate, this would now be added to the ArrayList. This is extensible, satisfying the Open/Closed Principle (OCP) as any number of destinations could be made available for any gate. This also required minimal changes from the previous implementation. The new implementation merely uses an ArrayList of Actions, where before one moveAction was used.

Refactoring Maps

The Lists of Strings which are used for the various maps which make up the game were extracted to the Map class in the utilities package. This enabled the Application class to be cleaned up, which is advantageous as it improves readability. The alternative is to leave the maps in the Application class, however, this has limited extensibility, as the Application class would need to keep track of too many things, therefore making the new solution better.

Refactoring Enemies Instantiation

The Factory Method Pattern was chosen to be implemented when instantiating new instances of enemies. This satisfied the Dependency Inversion Principle, as this implementation allows each of the enemy spawning grounds to be dependent on the Spawners interface, rather than a concrete implementation of this interface, which improves the extensibility of the system, as the different parts of the system remain independent.

An extra layer of abstraction is added to the system through this dependency inversion. Each enemy has an associated concrete factory, here referred to as spawners. These are all dependent on the Spawners interface, which provides one overloaded method, getEnemy, which can either take no parameters or the parameter for ‘weather’. In this way, any enemies which require extra parameters for instantiation are catered to. Similarly, a static overloaded method appears in the class, which takes a String, ‘enemyType’ and/or a ‘weather’ parameter.

By employing the Factory Method Pattern, the system also becomes more easily extensible, as any new enemies can be added in a very similar manner, with no dependency between the enemy’s spawning ground and the enemy itself.

An example interaction when creating a new Eldentree Guardian using this pattern was selected to be represented in the Interaction Diagram.

Additionally, whilst refactoring the system, it became clear that bosses should not be instantiated in the same manner, as bosses are added directly to game maps, rather than

through spawning. Therefore, a new abstract Boss class was created, extending from the Enemy class. This also implemented the Open/Closed Principle, as new Bosses can very easily extend from this Boss class, without requiring modifications to the existing system.

From the Assignment 1 sample solution, the Graveyard class is associated with the Spawner interface, with a HollowSoldierSpawner and presumably, a WanderingUndeadSpawner class implementing this interface. This solution is very similar to the solution that was employed, however, may use slightly less classes, as only one Graveyard class is employed, which likely passes the required enemy to the Spawners interface, as well as the probability of the enemy being spawned.

Using that solution, four new classes would need to be created, an Enemy concrete class and Spawner for each enemy and requiring reworking of the Hut and Bush classes from Assignment 2. However, this solution does not implement interfaces to describe what an enemy is capable of.

Refactoring Enemy Abilities

Two new interfaces, FollowCapable and MoveCapable were introduced, added to an abilities package, within the enemies package. These interfaces could be implemented by any enemies with those abilities, which meant less logic is processed in the Enemy class, an advantage which prevents the Enemy class from being a ‘god class’.

Each interface contains one method to be implemented in the relevant classes, by adding new behaviours to the enemy. This adheres to the Interface Segregation Principle, as no enemy implements an interface which they would not use.

Although this change means some classes may have the same playTurn method, which means the Don’t Repeat Yourself (DRY) principle is somewhat violated, this means the Enemy class is not a ‘god class’. Therefore, the advantages outweigh the disadvantages.

This is a much better implementation than the original one proposed for this assignment, which used checks of various enumerations in the Enemy class to add relevant behaviours. This is much more difficult to refactor as more enemies are added or new abilities are introduced for different enemies. Therefore, the current design is much better.