Design Rationale

By: MA_AppliedSession1_Group3

REQ 1: The Overgrown Sanctuary

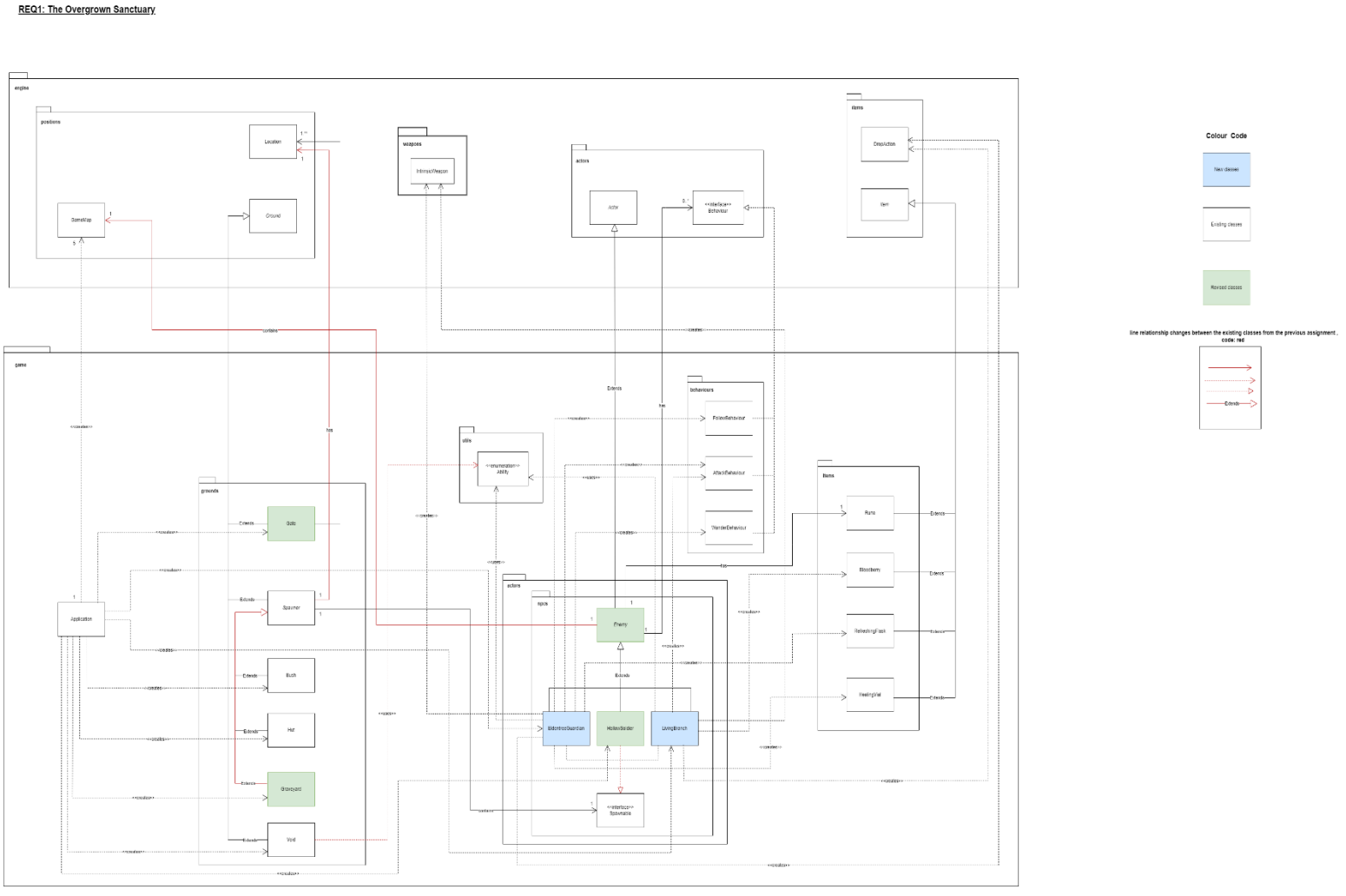

Figure 1: UML class diagram of REQ1 - The Overgrown Sanctuary

The provided diagram illustrates the entire system implementation for Requirement 1: The Overgrown Sanctuary. The primary goal of this design is to create a new game map known as the Overgrown Sanctuary, with a gate for accessing this new map. Within the Overgrown Sanctuary, the Bush and Hut are created, which will spawn new types of enemies, namely the Eldentree Guardian and Living Branch, at every turn of the game. These enemies will drop healing vials, refreshing flasks, or bloodberry items once they are defeated. Additionally, the Graveyard will be created to spawn Hollow Soldiers in the Overgrown Sanctuary. The Eldentree Guardian has the ability to wander around when the player is not nearby and can attack or follow the player until either the player or the enemy is unconscious if the player is nearby. In contrast, the Living Branch cannot wander and follow the player but can attack the player when nearby. This expansion of the system introduces new classes: the Eldentree Guardian class and the Living Branch class. These newly created classes will interact with the existing classes in the system.

As mentioned in the previous design rationale for REQ1: The Ancient Woods in Assignment 2, the abstract Spawner class has been introduced and will be extended by the Bush and Hut classes. In this requirement, the Graveyard class also extended the abstract Spawner class which will be discussed in the modification part. The functionality of the abstract Spawner class is further extended through an association relationship with the Location, which will be used in the later implementation of the game, which shown in the UML class diagram REQ1: The Overgrown Sanctuary. This design decision aligns with the Open/Closed Principle (OCP) and the Liskov Substitution Principle (LSP) that mentioned previously. Additionally, the abstract Spawner class has an association relationship with the Spawnable interface class to spawn different enemies at certain chances during each turn of the game. In this requirement, the abstract Spawner class will spawn instances of the EldentreeGuardian, LivingBranch, and HollowSoldier classes. This design choice reduces multiple dependencies (ReD) on various classes that may be implemented in the future, such as the EldentreeGuardian and LivingBranch classes, as well as existing classes like the HollowSoldier class. Furthermore, the existing Void class, which extends the abstract Ground class from the engine, has a dependency relationship with the Ability enum class. This is to check if every actor it encounters has the ability to step on it during each turn of the game, so the instance of the Void class can accurately eliminate the actor who does not have the ability to step on the Void. Alternatively, if the Ability enum class is not used, we would need to implement more if-else statements and instanceof operators (to check class types) in the Void class to ensure that all actors, regardless of whether they are the Player or any enemies, are eliminated. This would involve extra dependencies and violate the Open/Closed Principle (OCP) because when new actors are introduced, the existing code of the Void class would have to be modified to ensure that the condition of Void grounds eliminating all entities except the Eldentree Guardian and Living Branch holds true at all times.

To achieve the design goal, we introduced two new enemy classes into the extended system: EldentreeGuardian and LivingBranch. Both classes extend the existing Enemy abstract class mentioned in the previous Assignment 2 since they share common attributes and methods. This design choice aligns with the Don't Repeat Yourself (DRY) principle. In addition to adhering to DRY, this design also aligns with the Dependency Inversion Principle (DIP) and the Open/Closed Principle (OCP). The functionality of the abstract Enemy class is further extended through an association relationship with the GameMap, which will be used in the later implementation of the game, as shown in the UML class diagram for REQ5: A Dream? In this requirement, the abstract Enemy class still maintains an association relationship with the Behaviour interface class, as mentioned in the previous Assignment 2. However, the Behaviour varies due to the enemies introduced in the Overgrown Sanctuary. Since all enemies, whether Eldentree Guardian or Living Branch in the Overgrown Sanctuary, can attack the player, they share the same behaviour: attack behaviour. Therefore, the EldentreeGuardian class and

the LivingBranch class have a dependency relationship with the existing behaviour in the extended system, the AttackBehaviour class, which implements the Behaviour interface class from the engine. In contrast, the instance of the EldentreeGuardian class can wander around if the player is not nearby or follow the player if the player is nearby, we decided to create a dependency relationship with the existing behaviour classes in the extended system, the FollowBehaviour, and WanderBehaviour classes, both of which also implement the Behaviour interface from the engine. Additionally, the EldentreeGuardian class has a dependency relationship with the Ability enum class to prevent the Void class from affecting them. This is done by checking if every actor has the Ability.STEP_ON_VOID.

Furthermore, the EldentreeGuardian class creates a dependency relationship with the DropAction, HealingVial, and RefreshingFlask classes since they will drop those items once defeated. On the other hand, the LivingBranch class has a dependency relationship with the DropAction and Bloodberry classes, as these enemies will drop those items once defeated. Since all enemies regardless EldentreeGuardian and Living Branch in the Overgrown Sanctuary will drop Rune item when defeated, so the abstract Enemy class has an association relationship with the Rune class as mentioned in the previous Assignment 2.

Moreover, the Spawnable interface class, which will be implemented by LivingBranch and EldentreeGuardian classes also mentioned in the previous design rationale for REQ1: The Ancient Woods in Assignment 2. The existing subclass of the abstract Enemy class, HollowSoldier will also implement the Spawnable interface class, and this will discuss in the modification part. This design decision aligns with the Interface Segregation Principle (ISP), where the classes that implement this Spawnable interface only need interface methods. This approach also makes our system more extensible and maintainable.

The Application class creates (dependency relationship) a new GameMap class, Void class and Gate class, so the Application class has a dependency relationship with the GameMap class, Void class and Gate class. Besides, the Application class also creates (dependency relationship) the Bush class, Hut class, and Graveyard class as well as the instance of LivingBranch, EldentreeGuardian and HollowSoldier classes.

As we introduced abstract Spawner class instead of having Spawner interface class, this can reduce the code duplication (DRY) and dependency relationship (ReD), so we did not use the design given in the Assignment 1 sample solution. If we introduce interface Spawner class, we need to create multiple sub-classes related to the enemy’s types implementing the interface Spawner class. The new classes introduced may consists of code duplications, which violates the Don’t Repeat Yourself (DRY) principle. Besides, this will increase the system's complexity as many new classes are introduced as well as to increase the developers' difficulty if any errors occur during development.

Modification done on classes created in Assignment 2 for REQ 1

A few changes have been made to classes from the previous Assignment 2 interacting with new classes in the requirement 1, specifically, the Gate class, Graveyard class and HollowSoldier class.

In the last assignment, we had the Gate class with an association relationship with the Location class from the engine. The instance of the Gate class, which was dropped by Abxervyer (the boss) once defeated, could only lead to a single destination, back to the Ancient Woods. To make our system more extensible for future game implementations and to meet the requirement of having the gate dropped by Abxervyer lead to two destinations: back to the Ancient Woods and The Overgrown Sanctuary, we decided to modify the Gate class. We introduced a hash map to store multiple locations and loop through the map in the allowableActions method of the Gate class. The Gate class still maintains an association relationship with the Location class from the engine, but the multiplicity will be one or more instead of exactly one. The design decision adheres to the Open/Closed Principle (OCP) since the functionality of the Gate class is extended instead of modified.

All the grounds in the game that can spawn different enemies during each turn will extend this Spawner abstract class because they share common attributes and methods as mentioned in the previous design rationale for REQ1: The Ancient Woods in Assignment 2, so the Graveyard class will extend this Spawner abstract class as the Graveyard class also can spawn enemies. The Graveyard class will no longer from the Ground class from the engine directly. This design decision adheres to the Don't repeat Yourself (DRY) principle, as well as Liskov Substitution Principle (LSP), where a subclass can do anything its base class do and it does not disrupt the behaviour of our system.

We also decide to modify HollowSoldier class by implementing the Spawnable interface class since it can be spawned itself at every turn of the game as our main purpose of introducing Spawnable interface class in this extended system is to have Enemy subclasses that can spawn itself can implement it.

Furthermore, we refactored the Enemy abstract class and the HollowSoldier class in response to non-trivial usage of magic numbers in our previous implementation. Instead of directly instantiating variables with numeric values or passing numbers as arguments to methods, we opted to declare constant variables that store these numeric values. For example, in the case of the Hollow Soldier enemy, which has a 20% chance to drop a Healing Vial, we had a method named 'dropItemChance()' that required a percentage chance as an argument. Rather than passing the percentage directly, we declared a constant variable called 'DROP_HEALING_VIAL_PERCENTAGE' and used that as the argument for the 'dropItemChance()' method. Similar refinements were made when initializing the 'damage' variable in the HollowSoldier class and specifying key priorities in a hash map of behaviors within the Enemy abstract class.