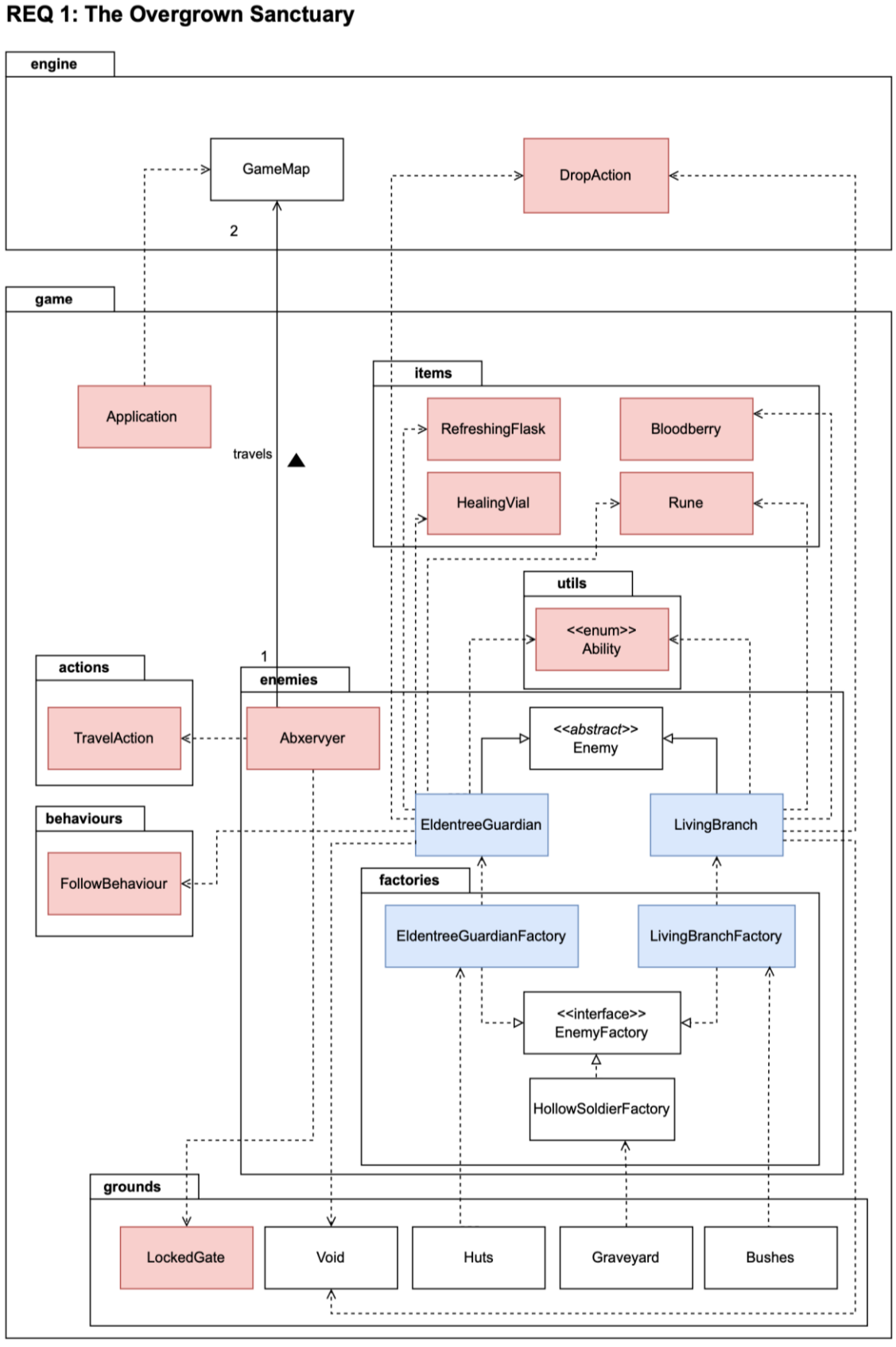
Design Rationale (REQ 1)

UML Diagram



Classes Added

- Class EldentreeGuardian extends Class Enemy
- Class EldentreeGuardianFactory implements Interface EnemyFactory
- Class LivingBranch extends Class Enemy
- Class LivingBranchFactory implements Interface EnemyFactory

Design Rationale

Classes EldentreeGuardian and LivingBranch

These two classes were created to represent two new types of enemies: Eldentree Guardian and Living Branch. They extend from the abstract **Enemy** class, which provides them with **AttackBehaviour** and **WanderBehaviour** upon instantiation. Since some enemies cannot follow the player, **FollowBehaviour** will only be added to the corresponsing enemy classes that can.

Based on the specification, we understand that the Living Branch cannot wander around or follow the player. However, it inherently has **WanderBehaviour** since it extends from the Enemy class. To remove **WanderBehaviour**, we override the constructor of the Living Branch class. For FollowBehaviour, it is added to the allowableActions of the **EldentreeGuardian** class, enabling it to follow the player.

The specification also mentions that the Eldentree Guardian can walk around in the Void without any consequences, while the Living Branch cannot be damaged by the Void since they are immune. We assigned a **CAN_BE_ON_VOID Ability** to both of these enemies so that they can stay freely on the void without being killed by it.

However, in our implementation, the Living Branch will not be spawned on top of the Void because we only allow it to spawn on top of bushes, not around them, as the Ed Forum reply indicated that spawning on top of a bush is acceptable. Because the Living Branch does not wander around, it will remain on top of the bush, with one bush spawning only one Living Branch enemy. While we are still giving the ability to be on the void to the Living Branch, possibly for future extensions, it is impossible for the Living Branch to be spawned on top of the Void in our current design.

Principles Followed

DRY (Don’t Repeat Yourself)

- The **Enemy** class effectively encapsulates common behaviour and attributes of enemies, preventing repetition in **EldentreeGuardian** and **LivingBranch** classes.
- Even though both the **EldentreeGuardian** and **LivingBranch** classes have a similar implementation of unconcious method, they both drop different amount of runes and also different items upon death. This is the reason why we did not refactor this method to reduce redundancy.

Single Responsibility Principle (SRP)

- Each class has a clear, single responsibility. **Enemy** handles common enemy behaviour, **EldentreeGuardian** handles behaviour specific to Eldentree Guardian enemies, and **LivingBranch** handles behaviour specific to Living Branch enemies.

Open-Closed Principle (OCP)

- Both classes are open for extension (can be subclassed to create variations) but closed for modification. Their core behaviours can be extended by overriding methods, but the core structure remains unchanged.

High Cohesion

- All classes show high cohesion, with methods and attributes closely related to the responsibility of the class.

Cons of the Design

- The current design may make it challenging to introduce new types of enemies with unique behaviours that don't fit the existing **Behaviour** framework. This is mentioned in the rationale part on top, where the abstract **Enemy** class assumes all of the enemies have **WanderBehaviour** and **AttackBehaviour** by nature, but we found that this is not the case for the Living Branch enemy. We did not modify this design because so far there is only one enemy with this exception. In future, this design might need to be modified, maybe only assign **AttackBehaviour** to every enemy by default, based on future requirements.

Interfaces EldentreeGuardianFactory and LivingBranchFactory

We had created two new factory classes, **EldentreeGuardianFactory** and **LivingBranchFactory**, to spawn two new types of enemies separately. Both of these classes implement the **EnemyFactory** interface, ensuring consistency and adherence to a common set of funtionalities. Essentially, the primary role of these factory classes is spawn the corresponding enemies.

To facilitate the spawning of enemies, spawnable grounds are designed to accept a factory class as a parameter, determining which enemy type they should generate. For instance, the code **new Bush(new LivingBranchFactory())** designates that Living Branch enemies are to be spawned on top of a bush.

The rationale behind this design pattern is to decouple the spawning logic from the enemy classes and confine it within the factory classes. This approach enhances modularity and ensures a clear separation of responsibilities. In scenarios where a **Bush** directly takes an instancce of the **LivingBranch** class as a parameter, there is a potential risk of exposing all public variables and methods of the LivingBranch class to the bush. However, the bush's sole requirement is to spawn the enemy, nothing more.

By externalizing the factory classes, we minimize the access that spawnable grounds have to the enemy classes. It not only securing the integrity of enemy classes but also simplifies the interaction between the spawnable grounds and the enemies.

Principles Followed

Single Responsibility Principle (SRP)

- The factory classes **EldentreeGuardianFactory** and **LivingBranchFactory** adhere to SRP as they have one reason to change: creating an instance of **EldentreeGuardian** and **LivingBranch**, respectively. The spawnable grounds, like the **Bush** class, also follow SRP by delegating the enemy creation responsibility to the factory classes.

Open-Closed Principle (OCP)

- The design adheres to OCP since new types of enemies and their corresponding factories can be added without altering the existing code. The **EnemyFactory** interface ensures that new factories can be created for new enemy types, promoting extensibility.

Interface Segregation Principle (ISP)

- The **EnemyFactory** interface is minimalist and specific, ensuring that the factory classes only need to implement the methods that are relevant to enemy creation. This avoids forcing classes to implement methods they do not use.

Low Coupling

- The design achieves low coupling. The spawnable grounds and enemy classes are loosely connected through the EnemyFactory interface, minimizing their dependencies on each other.

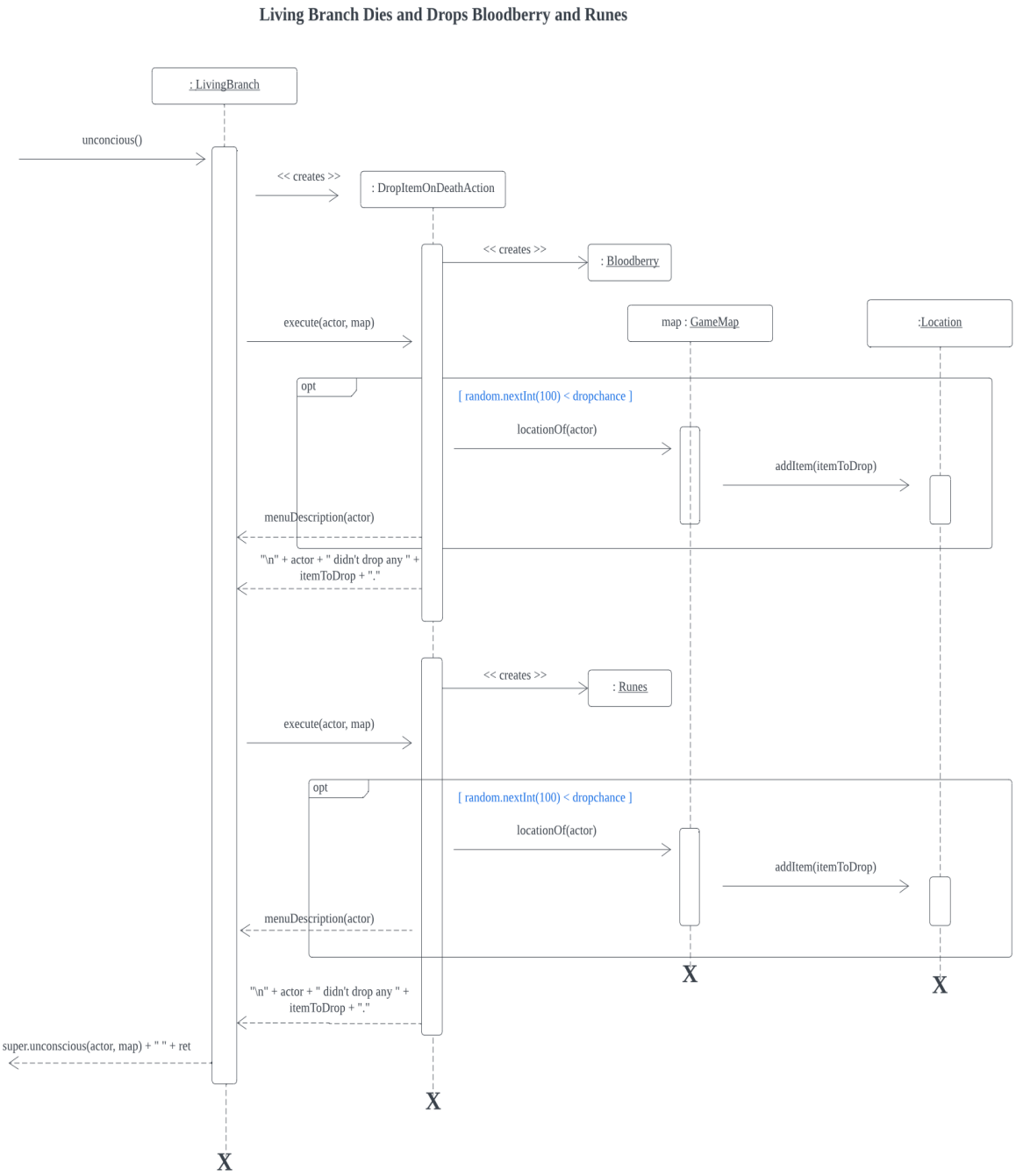
High Cohesion

- The classes demonstrate high cohesion as each class is focused on a single task. **Factory** classes are solely responsible for creating enemy instances, and the **Bush** class is only concerned with providing a ground for spawning.

Cons of the Design

- The design of factory classes might be overengineered for simplicity. If the instantiation of the enemies does not require any complex setup, then the factory pattern might be an unnecessarily complex solution. However, despite the apparennt complexity, sticking with this design offers several advantages. It future-proofs the application, as the enemy instantiation process might become more complex over time, requiring additional setup or dependencies. This is the reason why we stick to this implementation since Assignment 1.
- Introduction of factory classes might introduce some overhead, because each time an enemy is spawned, the associated factory class must be instantiated and its crteation method called, which might be less efficient than direct instantiation. However, since the design provide a clear separation of concerns, and encapsulating the enemy creation logic within specific classes adheres to SRP, we choose to stick to current implementation.

Sequence Diagram for Requirement 1



Comparison to the Assignment 1 Sample Solution

In the sample UML diagram provided for Assignment 1, a Spawner Interface is employed, wit various enemy spawners serving as its inheritors. For instance, the Wandering Undead enemy is spawned by a WanderingUndeadSpawner, which extends the Spawner class. This approach mirrors our group's implementation closely.

Addressing the Questions:

1. **How many classes would you need to create for this requirement if you used the Assignment 1 sample solution?**
   In our design, to incorporate a new enemy type, we need to define two classes: an EnemyFactory and a corresponding new Enemy class. This is consistent with the methodology outlined in the Assignment 1 sample solution.
2. **Do they contain some code duplications?**

   Based on the implementation code of Requirement 1, there is no significant code duplication within the EnemyFactory classes or the Enemy classes. Each factory class is responsible for instantiating a specific type of enemy, adhering to the SRP. Even though the code inside the enemy classes might seemed to have some duplication, like unconcious() method, and playTurn() method, each of the enemy classes have different logic for these, and hence, we do not consider this as code duplication.

3. **Can you think of a potential design alternative? What would be the cons of the alternative design?**

   A possible alternative design could be using a generic factory class or a factory method pattern, instead of separate factory classes for each enemy type. In this scenario, the factory could take a parameter or type argument to determine which enemy to instantiate.

   The alternative design has several cons. Firstly, it is less explicit, which means the alternative design might make the code less explicit, as it wouldn't be immediately clear from the class name which enemy type is being instantiated. Secondly, there might be a need for additional type checking or casting, which could introduce runtime errors if not handled correctly.

4. **What design did you end up using?**

   We opted for the design involving separate factory classes for each enemy type, as demonstrated in the provided code. The design choice promotes code readabiliy, maintainability, and ease of extension, aligning with SOLID design principles.