Design Rationale (REQ1)

Some major design changes/decisions

First of all, we have made some changes to the loot dropping mechanism. A interface of 'CanBeDroppedItem' is used instead of a class representing an item that can dropped. Any Item that can be dropped with a certain drop rate implements the above interface. This helps significantly reduce complexity to our code.

Secondly, the Gate class is modified so that it can take multiple Location instances as destinations. The destinations of the gate is stored within its capabilities list, meaning that it has the 'capability' to move to certain location. This is carefully implemented so that it is easily extendable to any maps in the future.

Furthermore, we also changed the name of 'ActorCanDrop' to 'Enemy' to further signifies the differences between a hostile enemy to actors of other kinds like the isolated traveller or player. The 'EldenTreeGuardian' and 'LivingBranch' class inherits from this class hence takes advantage of the code already implemented, enforcing code reusability.

Here are some of the design principles we used for this requirement.

Don't Repeat Yourself (DRY):
One of the fundamental principles in software engineering is to avoid duplicating code. In the extended game implementation, the design minimizes code duplication by reusing existing classes and introducing new classes where necessary, examples are such as 'EldenTreeGuardian' and 'LivingBranch' reusing the code implemented in the 'Enemy' class.

Open-Closed Principle (SOLID):
The design allows for easy extension of the game with new enemies and locations. For example, the Gate class has been extended to support multiple destinations. This ensures that future gates with multiple destinations can be added without modifying existing code.

Liskov Substitution Principle (SOLID):
The new enemy classes (Hollow Soldier, Eldentree Guardian, Living Branch) adhere to the Liskov Substitution Principle. They are subclasses of the Enemy class and can be used interchangeably with other enemies. This design supports the open-closed principle and allows for the addition of new enemy types without affecting existing code.

Code Smells and Connascence:
The design addresses code smells and connascence by encapsulating related functionality within classes and minimizing dependencies. For example, the Gate class encapsulates the logic related to unlocking and destination management, reducing the connascence between different parts of the code.

Advantages of the Design:

- The design is modular and extensible, making it easy to add new enemy types and locations in the future.
- Reusability is emphasized through the use of interfaces and abstract classes, reducing redundant code.
- The design minimizes code smells and connascence, making the codebase more maintainable and understandable.

Disadvantages and Possible Improvements:

- One potential disadvantage is that the game might become more complex as new enemy types and locations are introduced. To mitigate this, proper documentation and code organization are essential.
- While the design supports multiple destinations for gates, it does not explicitly enforce that destinations are distinct. Adding a check to ensure distinct destinations could be a future improvement.