Design Rationale

By: MA_AppliedSession1_Group3

REQ5: A Dream?

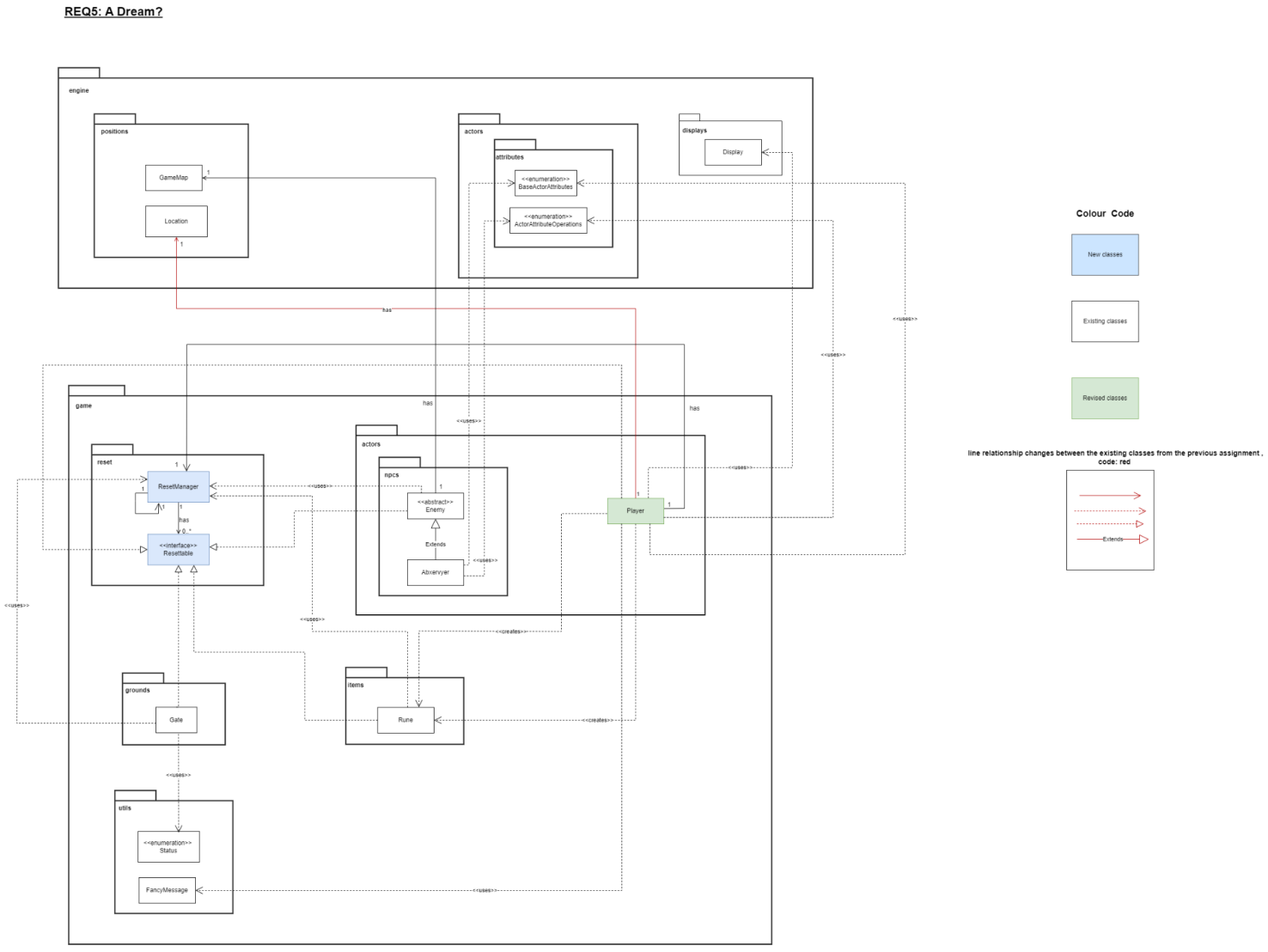


Figure 1: UML class diagram of REQ5 - A Dream?

The provided diagram shows the entire system implementation for Requirement 5, A Dream. The primary goal of this design is to allow resettable entities (Player, Gate, Enemy, Rune) to be reset upon the death of a player.

To achieve this goal of the design, we decided to introduce a new class, ResetManager. This is very similar to the WeatherManager class in Assignment 2. The ResetManager class is tasked with managing the reset process only. It keeps track of all entities that need to be reset upon the player's death and ensures that they are reset appropriately. By separating concerns in this way, we adhere to the **Single Responsibility Principle (SRP)**. There is a method run in ResetManager that reset all entities like Player, Enemy, Gate, and Rune. Along this line of thought, we know that the manager needs to be able to access all of the entities that will be reset. Therefore, we need an interface (Resettable, will be discussed later) and all entities that implement this interface are considered to be individuals that are resettable. Let the ResetManager class have an ArrayList attribute, a collection that holds all the entities that implement this interface. In this way, the reset manager will be able to access all resettable entities.

A new interface is introduced, which is Resettable. It can be used as an identifier to recognize entities that are resettable. It declares the reset() method, which must be implemented by any class that can be reset to a certain state when the player dies. The Resettable interface aligns with the **Interface Segregation Principle (ISP)** - to declare the interface/abstract method that resets the state of an object. Each implementing class defines its own way of resetting, ensuring separation of concerns. The interface supports the extension of the reset functionality. New classes can implement the Resettable interface to become resettable without altering the existing code. The ResetManager interacts with all resettable objects through this interface, ensuring it remains closed for modification. Therefore, it follow**s Open/Closed Principle (OCP**). It also follows **Liskov Substitution Principle (LSP)** - classes implementing the Resettable interface are substitutable for each other from the ResetManager's perspective. It treats all Resettable objects uniformly, invoking the reset() method without concerning the specific type of object. The ResetManager class depends on the abstraction (Resettable interface), not on the concretions (Player, Enemy, Rune, Gate). This decouples the ResetManager from the specific entity classes, making the system more flexible and easier to manage. This is an example of the Dependency **Inversion Principle(DIP).**

To fulfill the requirement 5, Enemy abstract class, Gate class, Rune class will implement the Resettable interface and override the reset() method since all the entities mentioned above are resettable when the game resets. We just need to write the individual corresponding reset rules inside the reset() method, for example, inside Gate's reset() we just need to add Status.LOCKED to indicate that Gate is reset (Gate is locked). All the enemies will be removed from all the game maps. All the enemies being removed from the game maps will also be removed from the resettable list. However, there is a little difference in implementation when resetting the rune. We added a new attribute 'reset', a Boolean value to the Rune class. It is set to false by default. It will be changed to true in the reset() method. When resetManager resets the rune, it simply turns the 'reset' boolean of the rune to true, and what really removes the rune is tick(), so the Rune class needs to override the tick method, and inside the body of the method is nothing more than determining the value of the 'reset', and if it's true, removing it from the Location's inventory. In this way, once the player dies, all runes on the ground will be removed (sometimes only a portion of the runes are removed after the player's death, with the rest taking a turn to be removed, this will be discussed in 2nd issue). All the issues will be discussed in the next section.

The player class will also implement the Resettable interface class**.** As the Player will be respawned right back where they started their journey, we need the Player initial location so we can utilise the moveActor method from the Location class to respawn the Player back to the building in the middle of the Abandoned Village. Hence, we decided to have the Player class with

an association relationship with the Location class from the engine by passing the instance of Location class as a parameter in the Player's constructor. Passing the instance of Location class can avoid the code length smells (bloaters). The values of the Player's attributes, such as HP and stamina,will be reset to full. Thus, it has dependency relationships with the ActorAttributeOperation and BaseActorAttributes enumeration classes. In the player's last location, a new rune item will be dropped, indicating the Player class creates (dependency relationship) a Rune item, thus the Player has a dependency relationship with the Rune class.

Modification done on Player class created in previous Assignment 1 for REQ5:

In the last assignment, we had the Player died due to any causes, the game would end, and a "YOU DIED" message was displayed. The implementation of the Player class' unconscious method was overridden by displaying the "YOU DIED" message only. To meet the requirement of resetting the Player based on the specifications, we decided to modify the Player class' unconscious method. In the modified unconscious method of the Player's class, we obtained the last location where the player stood and called the run() method of the ResetManager since we had an association relationship with the ResetManager class. As mentioned in the paragraph above, the Player class implements the Resettable interface class since the Player was resettable. The implementation in the reset method of Player class based on the specifications of requirement 5.

This design decision still follows the **Liskov Substitution Principle (LSP)**, where a subclass can do anything its base class does and it does not disrupt the behaviour of our system as the Player class substitutes the Resettable interface when the ResetManager invokes the reset method in the run() method. The Player class implementing the Resettable interface class tends to have a single purpose, which also aligns with the **Single Responsibility Principle (SRP)**.

Two issues have been identified and highlighted in Requirement 5, arising from limitations within the engine. These issues occur when displaying the game map after the player's character has died.

- ***When the player dies, it is not immediately visible on the map that the player is back at the respawn point.***
- ***The runes on the ground will not all be cleared at once because some of them are in front of where the player died.***

Before explaining the reason behind, let's remember the order of things the engine does in a turn:

1. print out the current map
2. tick all the maps (i.e., Location (ground + inventory) on all the maps)
3. player starts the turn first, then moves on to other actors

1st issue

- On the player's turn (1st turn in the game), when the player chooses to move to the void, this does not see any effect on "this turn" because the check that the player will be killed by the void is only executed on tick, and the player's turn occurs after the tick, so the player will not die on the current turn.
- Once the player moves to the void, it is the turn of the other actors, and when all the actors' turns are over, the engine will move on to the 2nd turn, which will print the map first.
- Note that at this point, you will see that the player is standing on top of the void in the map, which makes sense. As mentioned above, the only time a player can be killed by a void is when the void is ticking, and the tick in the 2nd turn occurs after printing the map, so the player is still standing on top of the void in the printed map.
- After printing the map, the engine goes to the tick step, and that's when the void kills the player, which is why we then see the YOUDIED message after the map is printed. After this, the engine goes to the player's turn (2nd turn in the game), so we will see a menu. Unlike normal cases, we don't see the current map. We can simply press 5 to let the player do nothing.
- The following images demonstrates the 1st display issue:

print the current map in the 1st turn

tick in the 1st turn

player's turn in the 1st turn

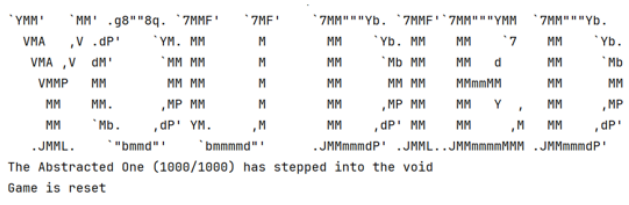
OPTIONS...

4

The Abstracted One (1000/1000) moves West

Other actors' turn...

print the current map in the 2nd turn



tick in the 2nd turn

```
The Abstracted One (1000/1000) has stepped into the void
Game is reset

The Abstracted One
HP: 1000/1000
Stamina: 2000/2000
Runes: 0
```

player's turn in the 2nd turn

OPTIONS...

5

The Abstracted One (1000/1000) does nothing

Other actors' turn...

print the current map in the 3rd turn

2nd issue

Continuing with the scenario above, when the player does nothing, we can see the new map where the player returns to the respawn location and the rune drops at the location where the player died. There is another issue in this scenario, let's now focus on the three runes that were dropped on the ground. Let's call the upper rune, rune1 and the middle rune, rune2 and the lower rune, rune3. After the player does nothing, the map in 3rd turn shows that only rune2 and rune3 has been removed.

- As said above, the condition for a rune to be removed is to reset first, so that its 'reset' value becomes true, and then tick. and reset will be called when the void's tick finds the player standing on top of it.
- tick calls are sequential (because of GameMap's The tick calls are sequential (because of GameMap's double for loop), so on the 2nd turn of the game, rune1 will call tick first, and nothing will happen because the player hasn't been killed by the void yet. Then void calls tick(), at which point the player dies and all the runes on the ground have their 'reset' set to true. then the tick is called on rune2, which is removed because it has a value of true, and the same applies to rune3. From this, we know why only rune2 and rune3 were removed from the printed map on 3rd turn.
- After printing the current map on the 3rd turn, engine will call tick for all maps, and this is where the rune1 has been removed. We can see this change in the printed map on 4th turn.

To further demonstrate this issue, I will use one more example with all the runes on the ground in locations that are after the location where the player died.

In this case, all runes can be successfully removed from the game when the player dies.