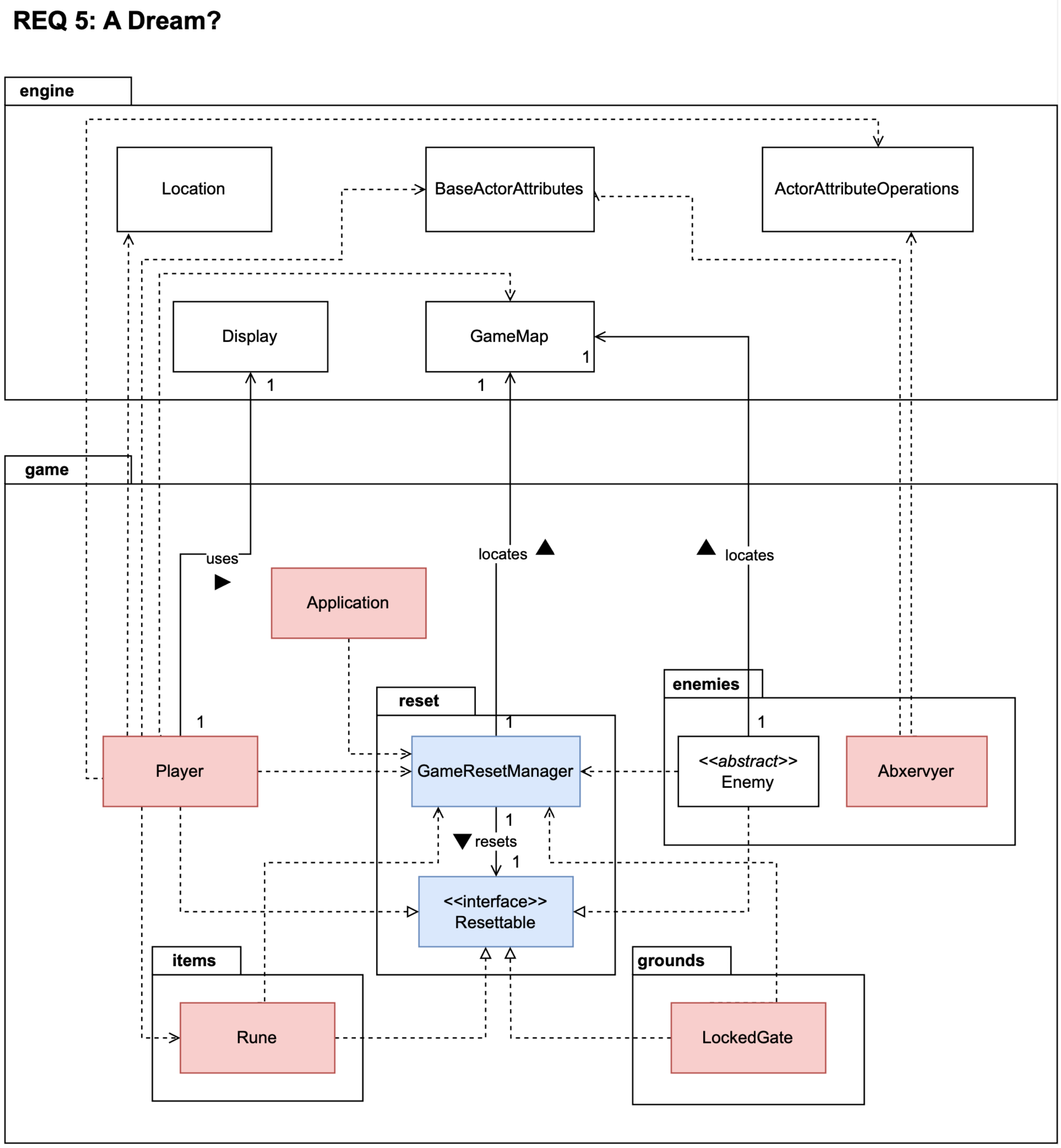
Design Rationale (REQ5)

UML Diagram



Classes Added

- Class GameResetManager
- Interface Resettable

Design Rationale

Interface Resettable

Following the specification for resetting the game after the player dies, we implemented the **Resettable** interface to handle this. The **Resettable** interface contains only one method, which is **reset**(). Everything that needs to be reset after the player dies will implement this interface.

Based on the requirement, the classes that implement the **Resettable** interface are **Abxervyer**, **Enemy**, **LockedGate**, **Rune** and **Player**. These are all the things that will be reset after the player dies. We understand that the **Abxervyer** class is a subclass of **Enemy**, but we still have It implement the **Resettable** interface. This is because the specification mentions that all spawned enemies (not including bosses) will be removed from the map, and bosses' health will be reset to full if they have not been defeated. To make this work, we have to override the **reset**() method in the **Abxervyer** class to define its own reset action.

The **reset**() action of **LockedGate** essentially removes its status of **UNLOCKED** and locks it again. The player can only unlock the gate again with an old key. For **Rune**, calling the **reset**() method assigns a **REMOVED** status to the **Rune** item. Inside the **tick**() function of the **Rune** class, whenever the **Rune** has the **REMOVED** status, the **Rune** will be removed from its current location.

Lastly, for the **Player** class, the **reset**() method reduces the balance in the player's wallet to 0, as well as sets the player's health and stamina to the maximum. All the other reset operations for the Player are handled inside the **unconcious**() method. The reason behind this design is that the unconcious method taks two classes as parameters (**Actor** and **GameMap**), while the **reset**() method takes no parameter. With the **GameMap** parameter, we can obtain the player's location on the map upon their death, remove the player from the map, and add a **Rune** item containing the player's initial balance at the place of the **Player** death. Hence, the reset() method only handles those operations that do not require the **GameMap** attributes.

Principles Followed

Single Responsibility Principle (SRP)

- The **Resettable** interface follows SRP as it has a single responsibilty: to reset the state of an object. Each class implementing Resettable defines its own reset behaviour, ensuring that a class has only one reason to change.

Open-Closed Principle (OCP)

- By using an interface, the design adheres to the OCP. New classes can implement the **Resettable** interface without modifyng exisyting code. This makes the system more extensible and maintainable.

Interface Segregation Principle (ISP)

- The **Resettable** interface is a good example of ISP because it doesn't force the implementing classes to use methods they don't need. It has only one method, **reset**(), ensuring that classes are not burdened with unnecessary methods.

Low Coupling

- The design promotes low coupling as classes are not directly dependent on each other. They depend on the **Resettable** interface, which reduces the dependencies between classes.

Cons of the Design

- There might be a minor code smell if the reset behaviour of a class is overly complex. This could violate the principle that an interface should represent a specific capability. However, since the current requirement for the reset operation of every class are pretty simple and straightforward, this should not be a problem.
- If a class implementing **Resettable** needs additional data or context to perform the reset, this design does not directly support it. A workaround would be required, possibly breaking some design principles. However, the design decision promotes simplicity and ensures that the reset behaviour remains decoupled from specific contexts or data sources. This maintains the separation of concerns and uphold the integrity of the interface.

Class GameResetManager

The **GameResetManager** is a manager that handles all the reset logic in the game upon a player's death. It maintains a list of **Resettable** classes, named resettables. Every class that implements the **Resettable** interface must register itself with the **GameResetManager** to be executed upon the player's death. This registration of resettable classes is done during their instantiation. A static final instance of **GameResetManager** is defined, ensuring a single instance of **GameResetManager** is available. This design pattern, known as the Singleton pattern, allows other classes to access this common instance, ensuring synchronization across the entire game.

The **Application** file, presumably the main class or configuration of the game, contains attributes of all the game maps. Consequently, the player's spawn map is passed as an attribute to the **GameResetManager**. This enables the **Player** class to obtain the map and respawn at the original spawn location. This logic is encapsulated within the **setSpawnMap** and **getSpawnMap** methods inside the **GameResetManager**.

To execute all the reset functions, the **execute**() method of **GameResetManager** is called inside the unconscious method of the **Player** class. Upon calling the **execute**() method, the **GameResetManager** iterates through every registered **Resettable** and invokes their **reset**() method sequentially. As a result, everything is reset back to its original state, in accordance with the specifications provided.

This design ensures that the reset logic is centralized and decoupled from the individual classes, promoting a clean and maintainable codebase. The use of the Singleton pattern for the **GameResetManager** ensures that there is a single point of control for the reset logic, facilitating easier management and coordination of the reset process across the game.

Principles Followed

Single Responsibility Principle (SRP)

- The **GameResetManager** has a single responsibility – to manage and execute the reset logic for all registered Resettable objects in the game. It ensures that all registered objects are rest upon a specific event, adhering to the principle.

Open-Closed Principle (OCP)

- The **GameResetManager** is open for extension but closed for modification. New **Resettable** classes can be added without changing the existing code in the **GameResetManager**, as they only need to implement the **Resettable** interface and register themselves.

Dependency Inversion Principle (DIP)

- **GameResetManager** depends on the abstraction (**Resettable** interface) and not on concrete implementatons, which is in line with the DIP principle.

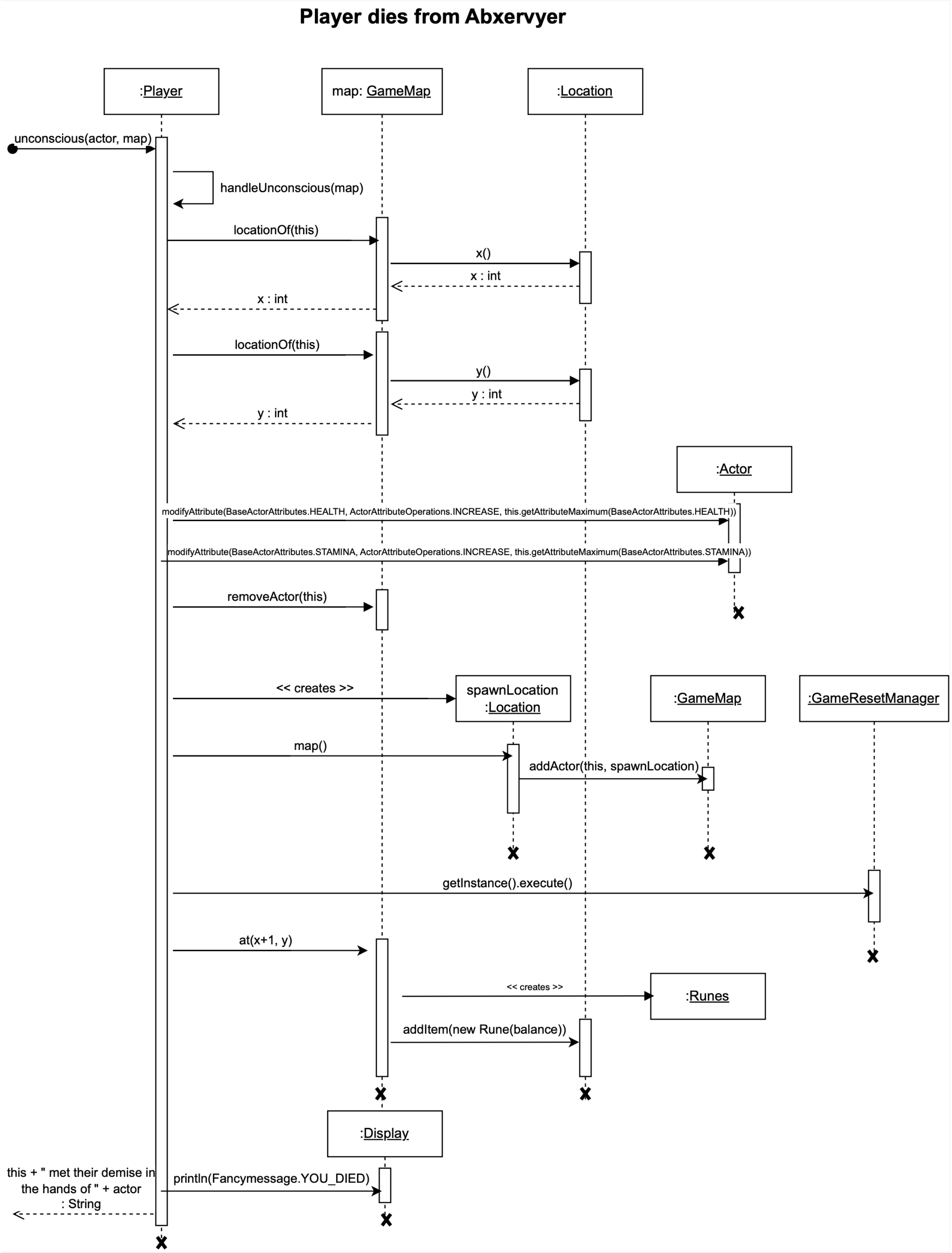
Singleton Design Pattern

- The **GameResetManager** is implemented as a Singleton, ensuring that there is only one instance of it in the system. This helps in coordinating actions and maintaining state consistency across the game.

Cons of the Design

- The design has a limited flexibility for **Resettable** Classes. If a **Resettable** class requires additional context or data to perform its reset, the current design does not directly support this. Each class has to handle this independently, possibly leading to repeated code and breaking the DRY principle. However, due to the simplicity of the current requirement, maintaining the current design is better since it follows most of the design principle.
- There is potential code smell where if the reset behaviour of a class is overly complex, it might lead to a minor code smell, as it could violate the printiple that an interface should represent a specific capability. However, this is mitigated by the simplicity and straightforwardness of the current reset operations.
- The Singleton pattern used for **GameResetManager** makes it a global state, which can make the system harder to debug and test. It also makes the system more susceptible to changes, as any change in the Singleton might have a ripple effect throughout the system. However, in our design, the **GameResetManager's** behaviour is stable and well-understood, and hence the risk of changes learning to issues might be lower than in a more distributed design.

Sequence Diagram for Requirement 5


Player dies from Abxervyer
:Player
map: GameMap
:Location
unconscious(actor, map)
handleUnconscious(map)
locationOf(this)
x()
x : int
x : int
locationOf(this)
y()
y : int
y : int
:Actor
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributeOperations.INCREASE, this.getAttributeMaximum(BaseActorAttributes.HEALTH))
modifyAttribute(BaseActorAttributes.STAMINA, ActorAttributeOperations.INCREASE, this.getAttributeMaximum(BaseActorAttributes.STAMINA))
removeActor(this)
<< creates >>
spawnLocation :Location
:GameMap
:GameResetManager
map()
addActor(this, spawnLocation)
getInstance().execute()
at(x+1, y)
<< creates >>
:Runes
addItem(new Rune(balance))
:Display
this + " met their demise in the hands of " + actor : String
println(Fancymessage.YOU_DIED)

Comparison to the Assignment 1 Sample Solution

All the new classes added here are not related to Assignment 1, and hence there is nothing to be compared here.

Addressing the Questions:

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

   A potential alternative design could be the implementation of the Observer pattern. In this scenario, **Resettable** classes would act as observers, and **GameResetManager** would be the subject.

   This is how it works. Each **Resettable** class would register itself with **GameResetManager** when instantiated, similar to the original design. Instead of a Singleton, **GameResetManager** could be instantiated and passed to each **Resettable** class through dependency injection. When the player dies, **GameResetManager** would notify all registered **Resettable** classes to perform their reset operations.

   There are a few cons of the alternative design. Firstly the design increased the complexity of code. The Observer pattern can increase the overall complexity of the system. Each **Resettable** class needs to iplement observer interfaces, and the **GameResetManager** needs to maintain a list of observers and handle their registration and notification. Moreover, if the observers are not properly unregistered, it can lead to memory leaks. Care must be taken to deregister **Resettable** objects when they are destroyed or no longer needed.

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

   The design that our group chosen incolves using a Singleton pattern for **GameResetManager**, with each **Resettable** class directly registering itself with the manager. This design centralizes reset logic, simplifies access to the reset manager, and ensures consistency across the system.