Design Rationale

By: MA_AppliedSession1_Group3

REQ 1: The Overgrown Sanctuary

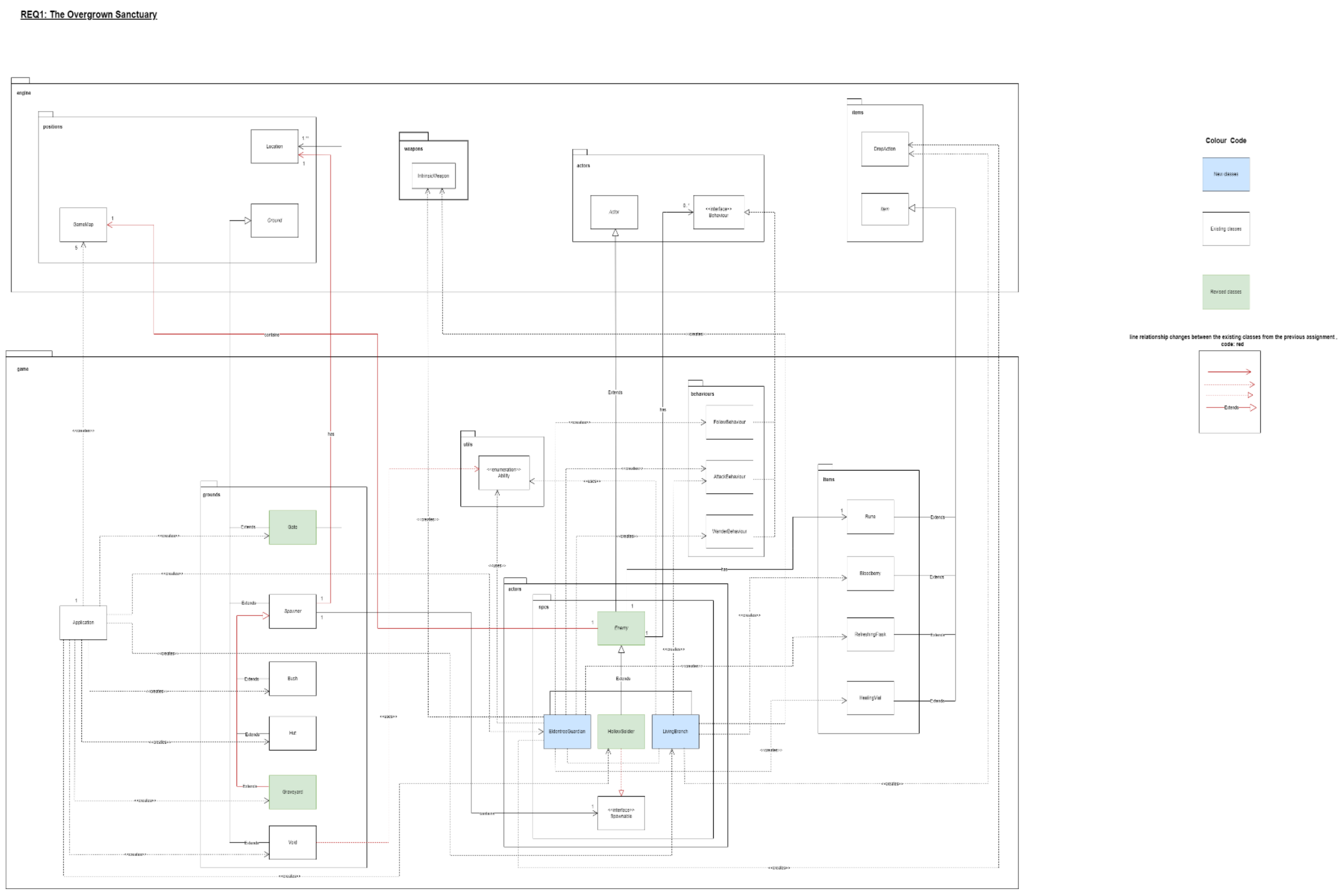


Figure 1: UML class diagram of REQ1 - The Overgrown Sanctuary

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.

<u>**Modification done on classes created in Assignment 2 for REQ 1**</u>

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.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 2: The Blacksmith</u>

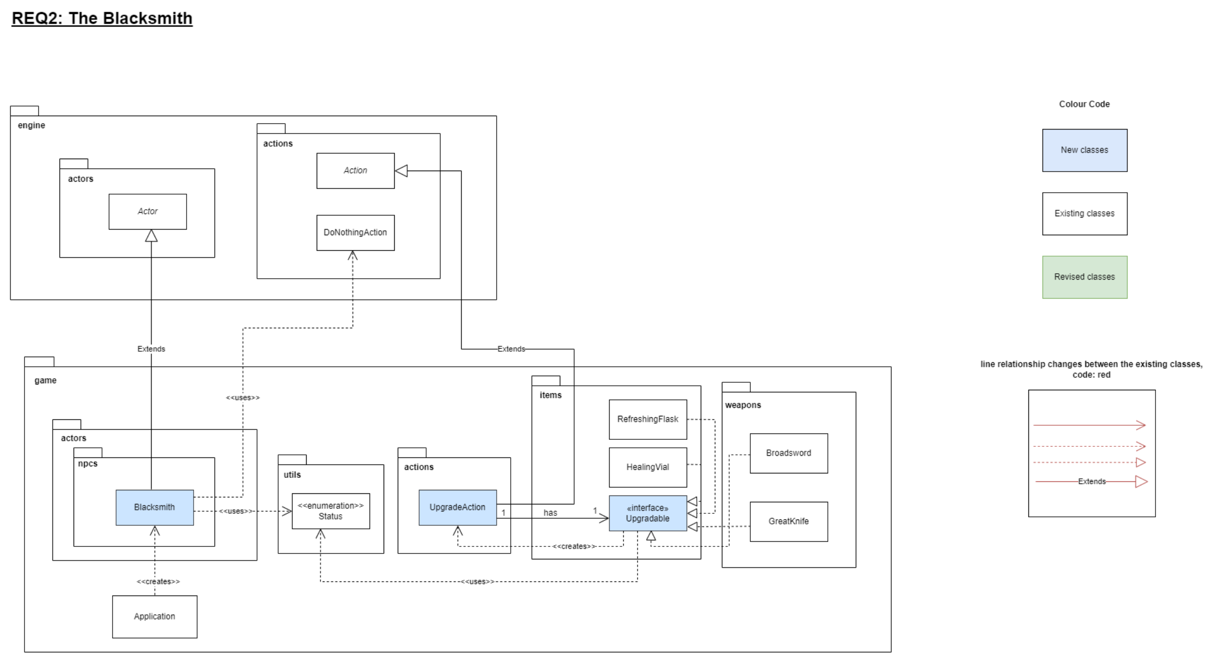


Figure 2: UML class diagram of REQ2 - The Blacksmith

The provided diagram shows the entire system implementation for Requirement 2, The Blacksmith. The primary goal of this design is to create an actor, The Blacksmith, that allows the players to upgrade the items in their item inventories with different prices. Thus, new classes and interfaces are introduced in the extended system: Blacksmith, UpgradeAction, and Upgradable. These classes will interact with the existing classes in the extended system.

To achieve the goal of the design, we created the Blacksmith class and the UpgradeAction class to execute their own responsibilities. This implementation follows the Single Responsibility Principle (SRP) because each class only focuses on their responsibilities. For instance, the UpgradeAction class only upgrades the items while not doing other work for the items. This makes the maintenance of the system easier as it would be more efficient to change the responsibility of a class without affecting the others.

Moreover, we decided to create an Upgradable interface in our extended system. This interface is implemented by the HealingVial, Broadsword, RefreshingFlask, and GreatKnife classes to make them upgradable. This design effectively aligns with the Don't Repeat Yourself (DRY) principle and prevents the code smell by using the polymorphism (Upgradable item) in the UpgradeAction class. Besides, the polymorphism with (Upgradable item) also obeys the Liskov Substitution Principle (LSP) and the Open/Close Principle (OCP) as these classes can execute the upgradedBy() methods in their own classes with different implementations, while do not affects the implementation of upgradedBy() in the Upgradable interface.

Additionally, the design aligns with the Interface Segregation Principle (ISP) and the Dependency Inversion Principle (DIP) with the creation of the Upgradable interface. For example, the GiantHammer does not implement the Upgradable interface as it cannot be upgraded, while some item and weapon classes can be upgraded with the implementation of the Upgradable interface, which follows the ISP. For DIP, the UpgradeAction class depends on the abstraction layer of Upgradable interface (item.upgradedBy) instead of the concrete classes such as the HealingVial class. Besides, with the private field of the “Upgradable item” in the UpgradeAction class, the encapsulation for the information hiding is considered.

On the other hand, the usage of the enumeration ‘Status.UPGRADE_PERSON’ and ‘Status.UPGRADED’ might be the drawback of the design as this will be a little touch of the code smell. However, the usage of the enumeration can replace the ‘instanceof’ operator which might add extra dependencies and violate the Open/Closed principle (OCP). Besides, the complexity of the classes would significantly increase if more related interfaces or abstract classes are added. Ultimately, we decided to continue with this current design.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ3: Conversation (Episode I)</u>

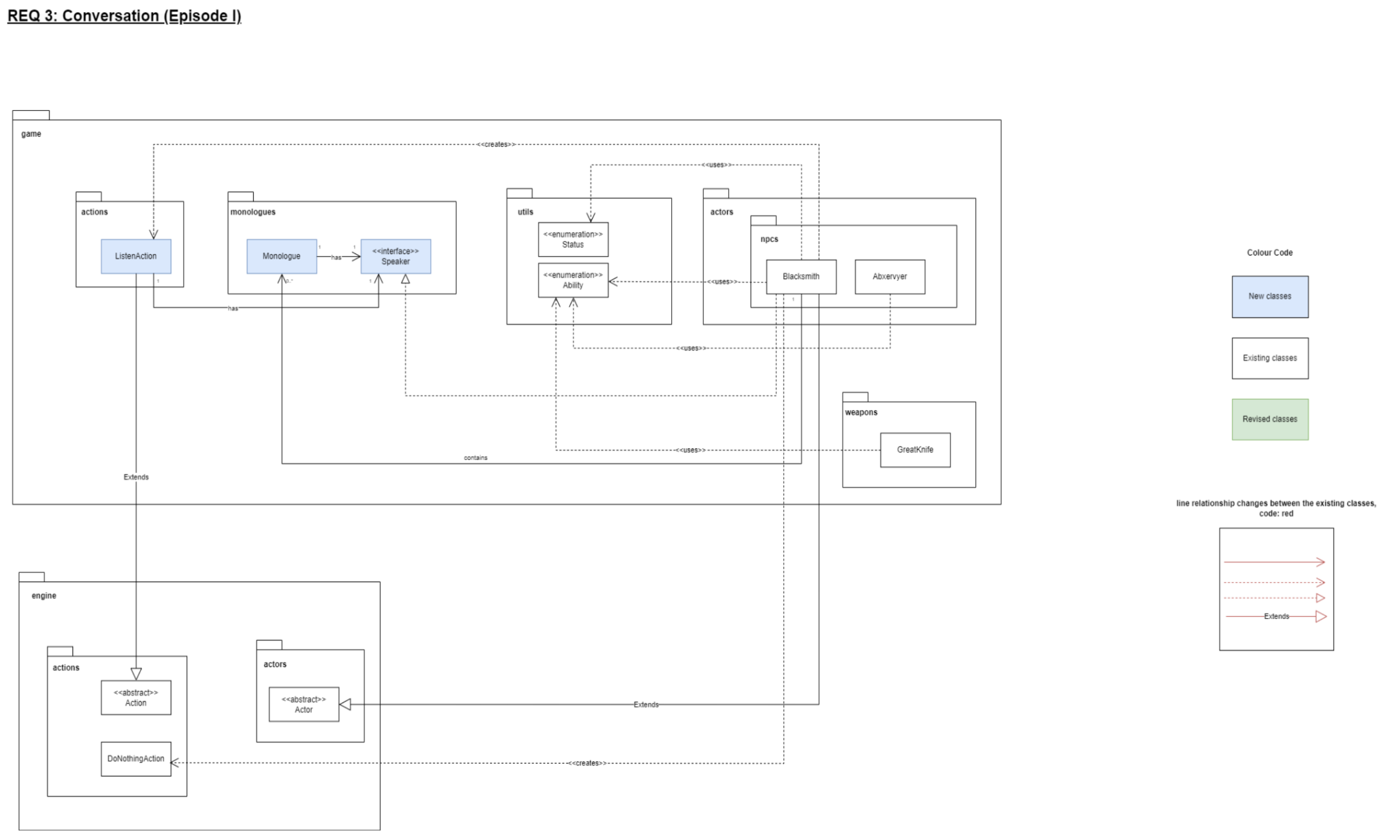


Figure 1: UML class diagram of REQ3 - Conversation (Episode I)

The provided diagram shows the entire system implementation for Requirement 3, Conversation (Episode I). The primary goal of this design is to allow the player to have the option to listen to monologue when the player is surrounded by Blacksmith. Additionally, the Blacksmith has a dynamic list of monologues and will randomly choose one to speak to the player.

To achieve this goal of the design, we decided to introduce a new interface class, Speaker. It can be used as an identifier to recognize actors who are capable of speaking monologue. It encapsulates the behavior of any game entity that can produce a monologue. The Blacksmith class implements the Speaker interface. Since it is substituting the Speaker interface, it adheres to the **Liskov Substitution Principle (LSP)**. This means that any instance of Blacksmith can be used wherever a Speaker is expected without issues. The Speaker interface is open for extension (any new actor can implement it to become a speaker) but closed for modification. If a new speaker is introduced, the existing code does not need to change. This design adheres to the **Open Closed Principle** (**OCP**). Besides, any newly introduced Actor classes who are also a Speaker can implement the Speaker interface. The Speaker interface is a clear example of **Interface Segregation Principle(ISP)**. It contains interface methods for any speaker entity to implement. If there were more behaviors to be added, they should be added in separate, targeted interfaces to ensure classes implement only what they need.

At the same time, the Blacksmith class has an association relationship with the Monologue class to store all the monologues that can be spoken. Therefore, we also introduced a new Monologue class. The primary reason for creating this Monologue class is to serve as a wrapper class for storing String conversations. Having the Monologue wrapper class encapsulates all the information related to a monologue that can be spoken by an Actor, in this current implementation, the Blacksmith. This information includes the text of the monologue, the conditions for delivering it, and the speaker. Using a plain String would scatter these different pieces of information throughout your codebase, making it harder to manage, read, and maintain. In contrast, a Monologue object encapsulates these details neatly. For example, we can store the conditions under which a monologue is available for the Blacksmith to speak, ensuring that the list of monologues in the blacksmith's store remains dynamic. When generating the list of monologues, only those monologues that evaluate to true are added to the array list. As the game evolves, we might need to add more attributes or behaviors related to monologues. For instance, we may want to track who has heard a specific monologue, the date it was delivered, or any special effects associated with certain monologues. With a dedicated Monologue class, we can easily extend it to include new attributes or methods without modifying the sections of your code that utilize monologues. This adheres to the **Open/Closed Principle (OCP)** of SOLID, which encourages extending behavior without modifying existing code. The Monologue Class also follows the **Single Responsibility Principle (SRP)**. It is responsible for encapsulating a single monologue and its conditions. Its primary purpose is to represent a monologue and its related properties. Hence, we decided to create this Monologue wrapper class instead of using plain String as our design choice for this requirement.

Another new class introduced is the ListenAction class, which directly extends the Action abstract class from the engine. This action is responsible for fetching a list of possible monologues, filtering them based on their conditions, and ultimately delivering a random monologue from the eligible ones. This class follows the **Single Responsibility Principle (SRP)** because it is solely responsible for executing the listening action and fetching the relevant monologue. The ListenAction class depends on the Speaker interface rather than a concrete implementation. This decoupling adheres to the **Dependency Inversion Principle (DIP)**, providing flexibility in terms of which speakers can be listened to without tightly coupling the action to specific speaker classes. Moreover, the ListenAction class relies on the abstractions layer (Speaker interface) rather than concrete implementations. This design choice ensures that any changes made in concrete implementations (concrete classes) will not affect the abstractions layer.

Another critical aspect of this requirement is determining whether the player has defeated the boss or is holding the Great Knife. This requirement involves checking if the player has the 'STAB_AND_STEP' ability when they have the Great Knife in their inventory, and adding the 'DEFEATED_ABXERVYER' capability when the boss is defeated. To accomplish this, we use the Ability enumeration class to differentiate between the abilities of items carried and the status of the boss being defeated. This allows us to easily filter out monologues using the built-in 'filter()' method of ArrayList based on the player's current abilities. Using the Ability enumeration class eliminates the need for using 'instanceof' (checking class type), which can potentially violate the Liskov Substitution Principle (LSP) when using the 'instanceof' operator or 'object.getClass().getName()' to identify the actual subclass. However, this design decision has a drawback. To identify the presence of monologue choices in the list of the Speaker interface class, whether the player is carrying the Great Knife or has defeated Abxervyer, we need to hardcode these abilities to fulfill the scenario, which introduces a minor code smell. Nonetheless, our design decision aligns with the Open/Closed Principle (OCP) as we use the built-in 'filter()' method, which avoids the need for if-else statements and modifications to the ListenAction class, thus upholding the Open/Closed Principle (OCP).

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ4: Conversation (Episode II)</u>

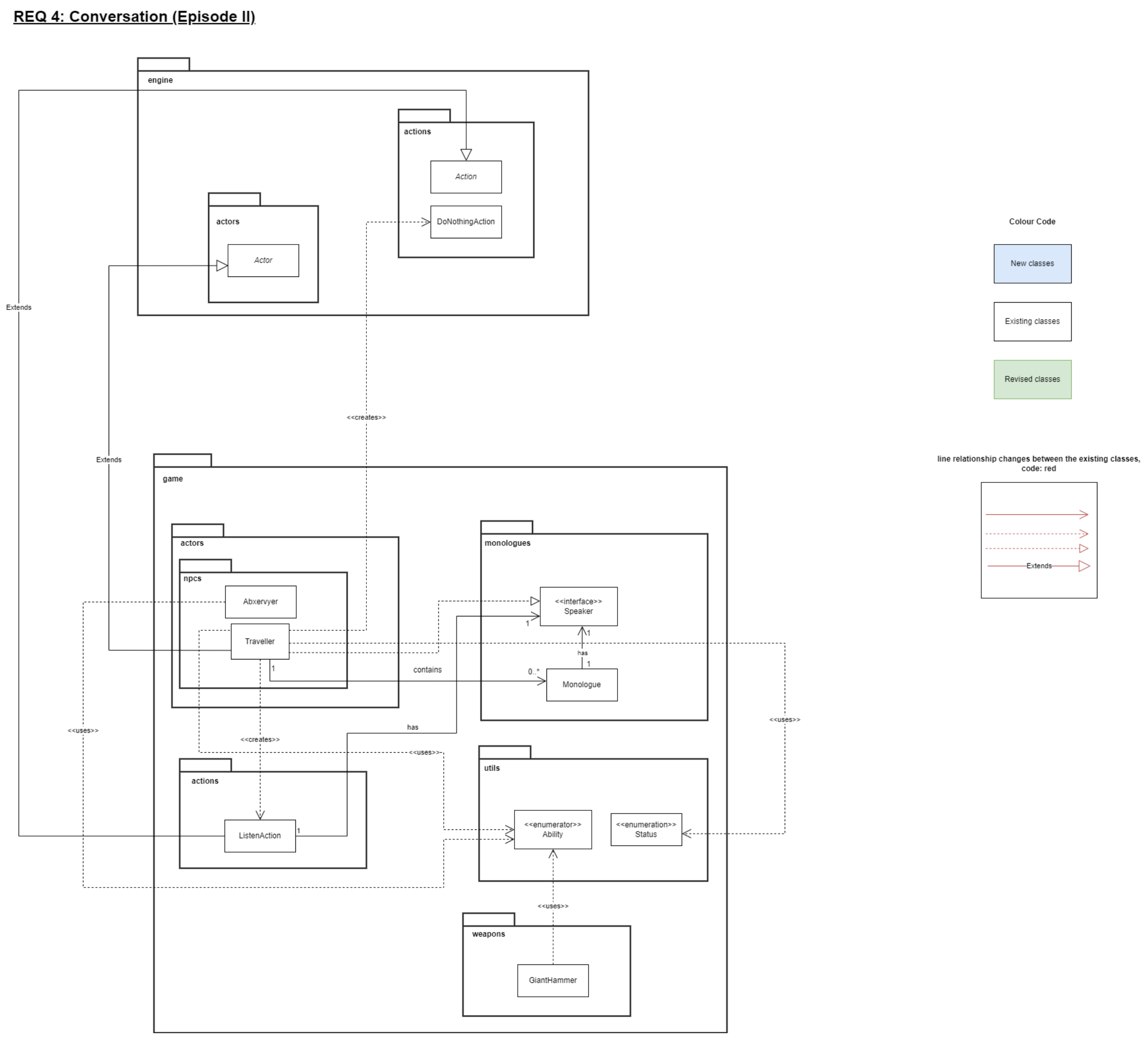


Figure 1: UML class diagram of REQ4: Conversation (Episode II)

The provided diagram shows the entire system implementation for Requirement 4, Conversation (Episode II). The primary goal of this design is to enable the player to listen to the monologue of the isolated traveller found in the Ancient Woods. Some monologue options can only be performed when certain conditions are met. The design rationale is similar to the Requirement 3 because the functionality is a continuation of requirement3.

To achieve this design goal, we introduced a new interface class called "Speaker." This interface serves as a marker to identify characters capable of delivering monologues within the game. It abstracts the behaviour of any in-game entity that can engage in monologue conversations.The Traveller class was modified to implement the Speaker interface. By doing so, it complies with the **Liskov Substitution Principle (LSP**), ensuring that any instance of the Traveller can be used in situations where a Speaker is expected without causing issues. The Speaker interface is designed to be open for extension, meaning that it can be easily implemented by new characters or entities to enable them to participate in monologue conversations. Additionally, it follows the **Open-Closed Principle (OCP**), which means that it remains closed for modification, ensuring that the existing code doesn't need to change when new speakers are introduced. The Speaker interface is a prime example of adhering to the **Interface Segregation Principle (ISP)**. It features a single method that any speaker entity must implement, ensuring that classes are only obligated to provide the specific behaviour relevant to their role. If additional behaviours need to be incorporated in the future, they should be introduced through separate, targeted interfaces to maintain the principle of providing classes with only the functionality they require.

At the same time, the Traveller class needs to have an ArrayList attribute to store all the monologues that can be spoken (This is one modification). We also introduced a new class Monologue. We used it as a wrapper for the monologue string, it encapsulates all the information related to a monologue in one place. It includes the text of the monologue, the condition for delivering it (boolean), and the speaker. Using a plain String would mean that these different pieces of information would be scattered throughout your codebase, making it harder to manage, read, and maintain. However, a Monologue object encapsulates these details neatly. For example, we can store the condition that a monologue is available for Traveller to speak, thus ensuring that the list of monologues in the Traveller's store is dynamic. When generating the list of monologues, only those monologues that conditions are true are added to the array list. As your game evolves, you might need to add more attributes or behaviours related to monologues. For example, you might want to track who has heard a specific monologue, the date it was delivered, or any special effects associated with certain monologues. With a dedicated Monologue class, you can easily extend it to include new attributes or methods without modifying the places in your code that use monologues (use actor.hasCapability() to determine whether the condition is true or false). This adheres to the **Open/Closed Principle (OCP)** of SOLID, which encourages extending behaviour without modifying existing code. Monologue Class follows the **Single Responsibility Principle (SRP)** well. It is responsible for encapsulating a single monologue and its conditions. Its primary purpose is to represent a monologue and its related properties.

There is one more new class, ListenAction class. This action fetches a list of possible monologues, filters them based on their conditions, and finally delivers a random monologue from the eligible ones. This class follows **Single Responsibility Principle (SRP)** because it is solely responsible for executing the listening action and fetching the relevant monologue. The ListenAction class depends on the Speaker interface rather than a concrete implementation. This decoupling adheres to the **Dependency Inversion Principle (DIP**), as it allows for flexibility in terms of which speakers can be listened to without tightly coupling the action to specific speaker classes.

Another important part of this requirement is how to determine if the player has defeated the boss or if the player is holding the Giant Hammer (This is another modification). Previously, we had a SKILL object in our Ability enumeration class, which was meant to indicate that a weapon had a special ability. Instead of giving GiantHammer a SKILL, we can just indicate the specific skill of this weapon, thus we will give GiantHammer, the weapon, a GREAT_SLAM ability. Doing so would fulfil the requirement of checking if the player is holding Giant Hammer (If the player has Giant Hammer in his inventory, the player has the GREAT_SLAM ability). The same logic can be applied to the problem of "checking if the boss is defeated", when the boss is defeated, just add a DEFEATED_ABXERVYER capability to the player. For the case “Once the player defeats Abxervyer & they still hold the giant hammer”, use & boolean operation to ensure that the Monologue is only added once the two conditions are true at the same time.

Another critical aspect of this requirement is determining whether the player has defeated the boss or is holding the Giant Hammer. This requirement involves checking if the player has the 'GREAT_SLAM' ability when they have the Giant Hammer in their inventory, and adding the 'DEFEATED_ABXERVYER' capability when the boss is defeated. To accomplish this, we use the Ability enumeration class to differentiate between the abilities of items carried and the status of the boss being defeated. This allows us to easily filter out monologues using the built-in 'filter()' method of ArrayList based on the player's current abilities. Using the Ability enumeration class eliminates the need for using 'instanceof' (checking class type), which can potentially violate the **Liskov Substitution Principle (LSP)** when using the 'instanceof' operator or 'object.getClass().getName()' to identify the actual subclass. However, this design decision has a drawback. To identify the presence of monologue choices in the list of the Speaker interface class, whether the player is carrying the Giant Hammer or has defeated Abxervyer, we need to hardcode these abilities to fulfill the scenario, which introduces a minor code smell. Nonetheless, our design decision aligns with the **Open/Closed Principle (OCP**) as we use the built-in 'filter()' method, which avoids the need for if-else statements and modifications to the ListenAction class, thus upholding the **Open/Closed Principle (OCP)**.

Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ5: A Dream?</u>

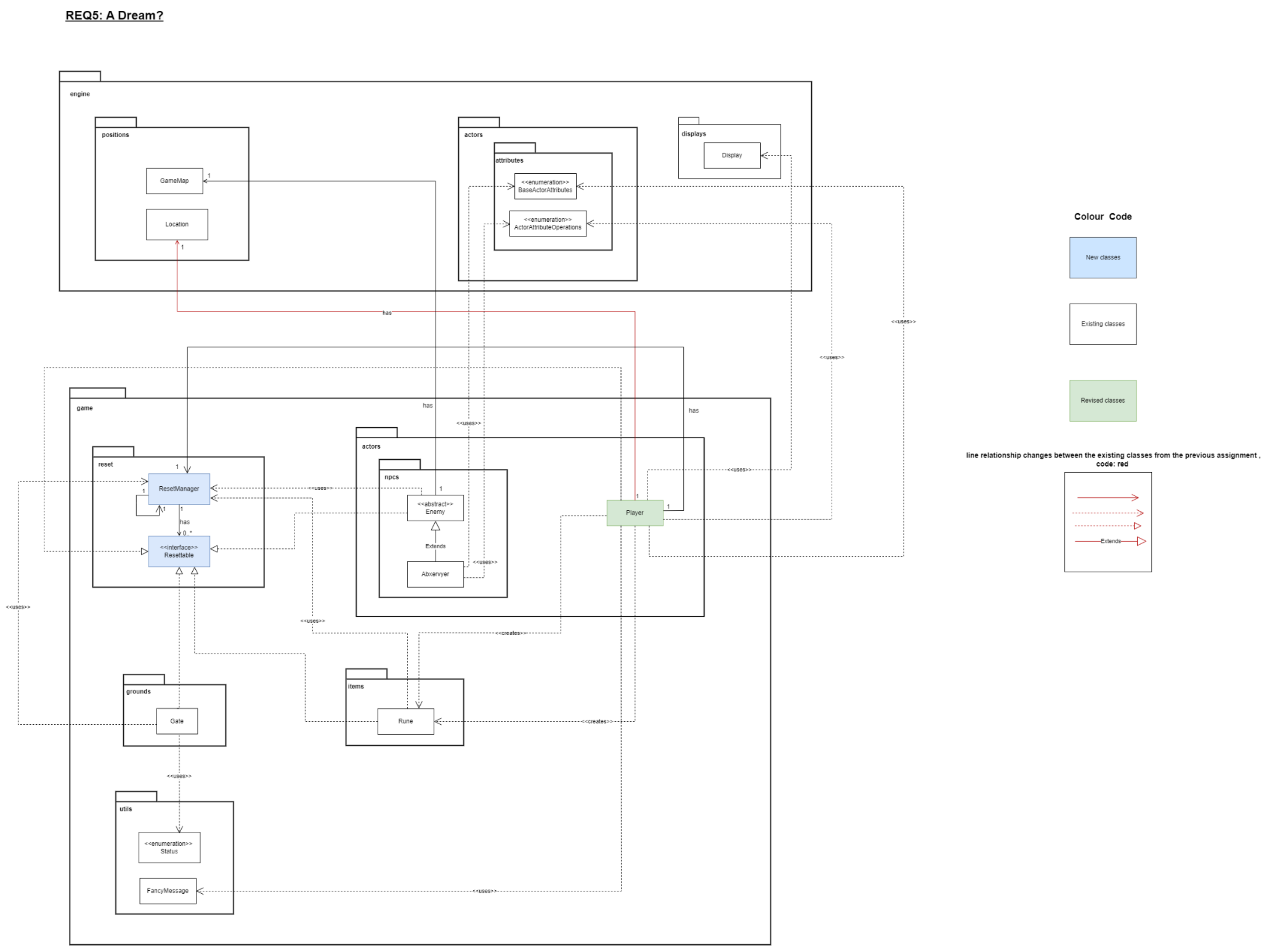


Figure 1: UML class diagram of REQ5 - A Dream?

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:

```
..............................=....................t.......
...#######........................................n........
...#__......................................++++..t........
...#..___#..................................++++++t.....t..
...###.###................#######.............+++..........
..........................#___B_#...............+++........
........~~................#1____#................+.........
.........~~~..............###_###...............++.........
...~~~~~~~~....+++.........................................
....~~~~~........+++++++.........$........###..##...++++...
~~~~~~~..............+++@..........n......#___..#...++.....
~~~~~~.................++.........$.......#..___#....+++...
~~~~~~~~~............................$....#######.......++.
```

print the current map in the 1st turn

tick in the 1st turn

OPTIONS...

player's turn in the 1st turn

4

The Abstracted One (1000/1000) moves West

Other actors' turn...

```
..............................=...................t........
...#######........................................n........
...#__......................................++++.........t.
...#..___#..................................+++++++t.......
...###.###................#######.............+++..........
..........................#___B_#...............+++........
........~~................#1____#................+.........
.........~~~..............###_###...............++.........
...~~~~~~~~....+++.........................................
....~~~~~........+++++++.........$........###..##...++++...
~~~~~~~..............++@...........n......#___..#...++.....
~~~~~~.................++.........$.......#..___#....+++...
~~~~~~~~~............................$....#######.......++.
```

print the current map in the 2nd turn

```
`YMM'   `MM'  .g8""8q. `7MMF'   `7MF'    `7MM"""Yb. `7MMF'`7MM"""YMM  `7MM"""Yb.
  VMA   ,V  .dP'    `YM. MM       M        MM    `Yb. MM    MM    `7    MM    `Yb.
   VMA ,V   dM'      `MM MM       M        MM     `Mb MM    MM   d      MM     `Mb
    VMMP    MM        MM MM       M        MM      MM MM    MMmmMM      MM      MM
     MM     MM.      ,MP MM       M        MM     ,MP MM    MM   Y  ,   MM     ,MP
     MM     `Mb.    ,dP' YM.     ,M        MM    ,dP' MM    MM     ,M   MM    ,dP'
   .JMML.     `"bmmd"'    `bmmmmd"'      .JMMmmmdP' .JMML..JMMmmmmMMM .JMMmmmdP'
The Abstracted One (1000/1000) has stepped into the void
Game is reset

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

tick in the 2nd turn

OPTIONS...

player's turn in the 2nd turn

5

The Abstracted One (1000/1000) does nothing

Other actors' turn...

```
..............................=............................
...#######........................................n........
...#__......................................++++...........
...#..___#..................................+++++++........
...###.###................#######.............+++..........
..........................#__@B_#...............+++........
........~~................#1____#................+.........
.........~~~..............###_###...............++.........
...~~~~~~~~....+++.........................................
....~~~~~........+++++++.........$........###..##...++++...
~~~~~~~..............++$...........n......#___..#...++.....
~~~~~~.................++.................#..___#....+++...
~~~~~~~~~.................................#######.......++.
```

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.

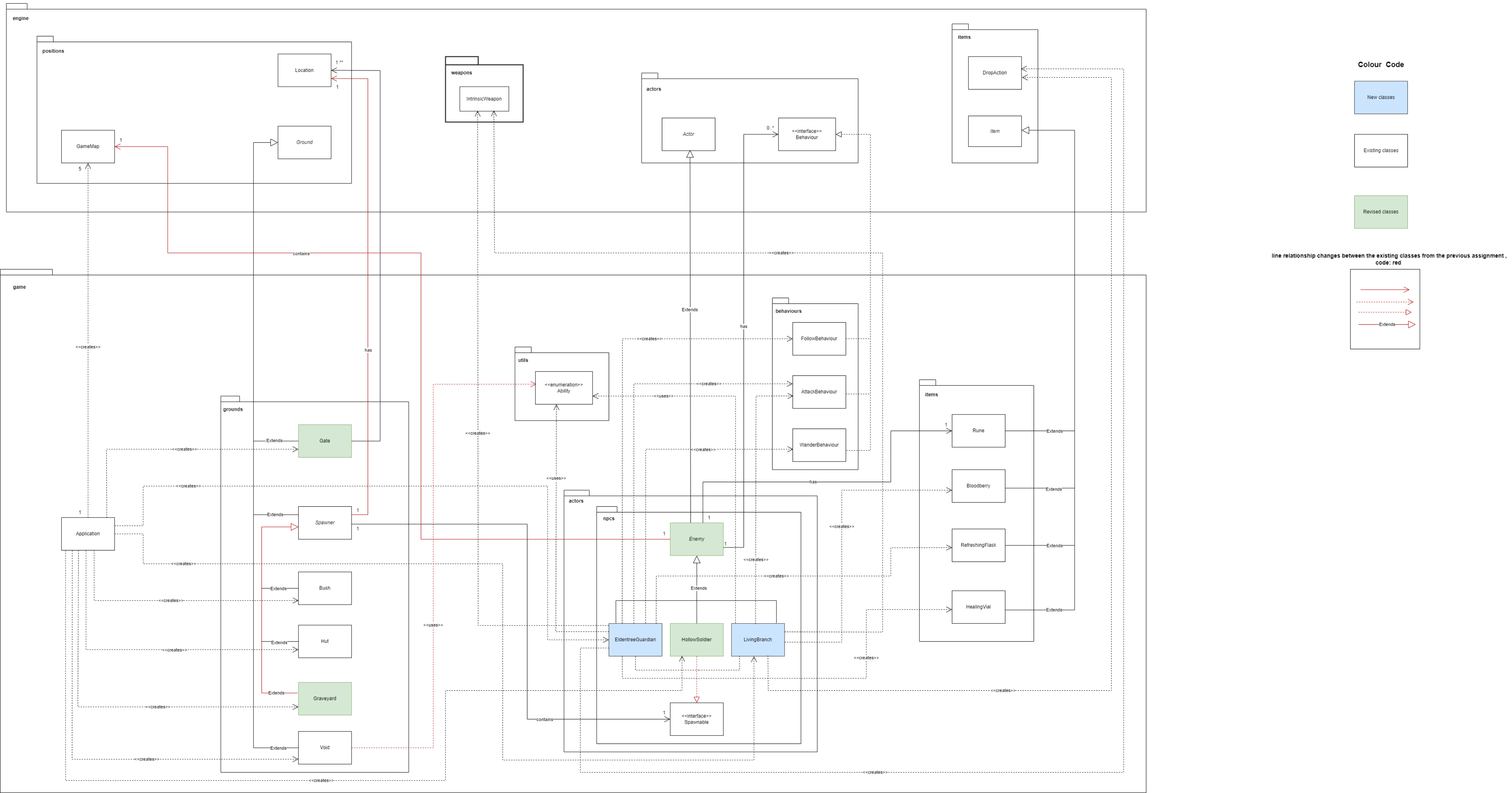
```
..............................=............................
...#######........................................n........
...#__......................................++++...........
...#..___#..................................+++++++........
...###.###................#######.............+++..........
..........................#__@B_#...............+++........
........~~................#1____#................+.........
.........~~~..............###_###...............++.........
...~~~~~~~~....+++.........................................
....~~~~~........+++++++..................###..##...++++...
~~~~~~~..............++$...........n......#___..#...++.....
~~~~~~.................++.................#..___#....+++...
~~~~~~~~~.................................#######.......++.
```

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.

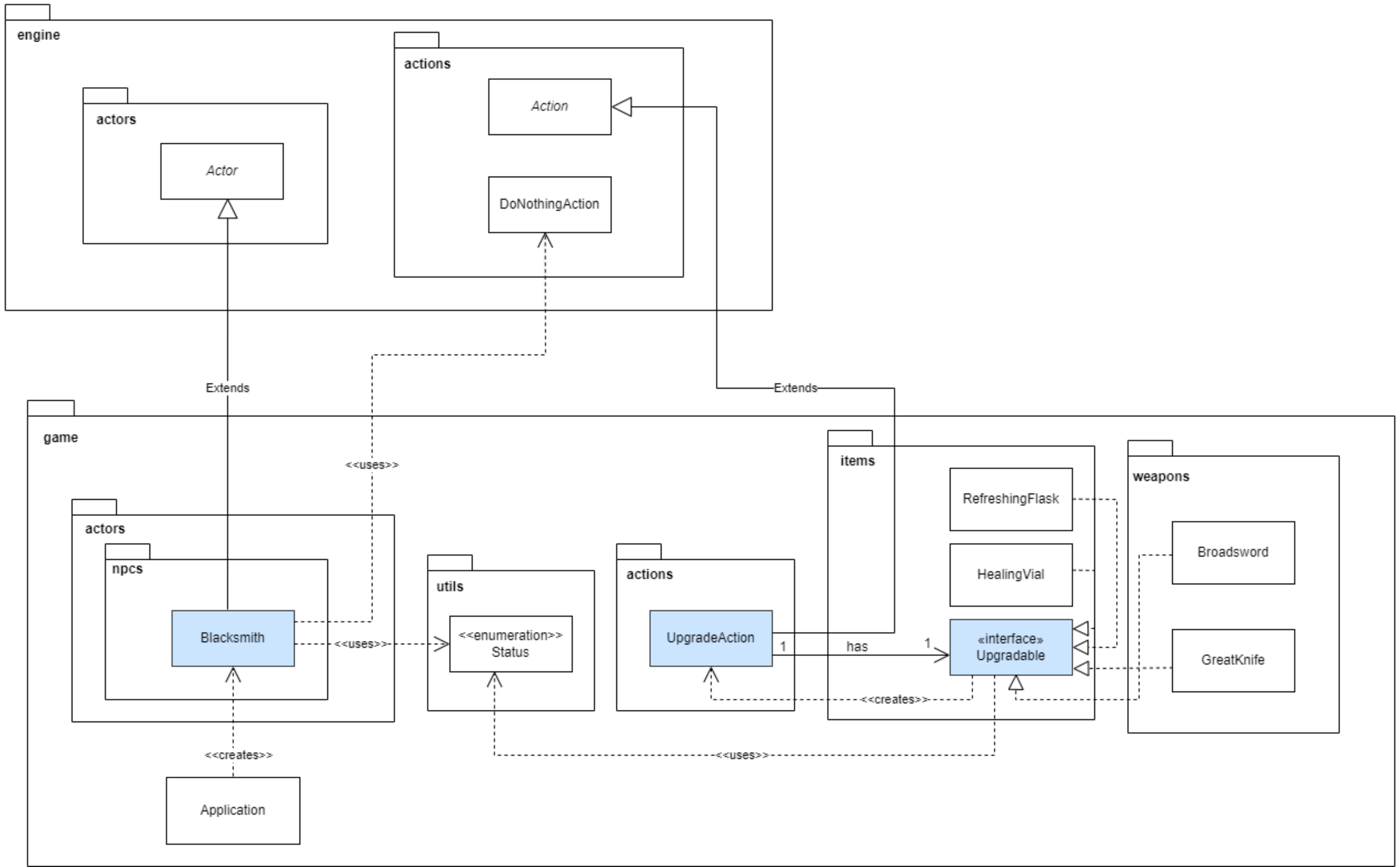
```
..............................=............................  ..............................=............................
...#######........................................tt.......  ...#######........................................n........
...#__......................................++++...........  ...#__......................................++++...........
...#..___#..................................++++++t........  ...#..___#..................................+++++++........
...###.###................#######.............+++..........  ...###.###................#######.............+++..........
..........................#___B_#...............+++........  ..........................#__@B_#...............+++........
........~~................#1____#................+.........  ........~~................#1____#................+.........
.........~~~..............###_###...............++.........  .........~~~..............###_###...............++.........
...~~~~~~~~....+++................t........................  ...~~~~~~~~....+++.........................................
....~~~~~........+++++++........t.........###..##...++++...  ....~~~~~........+++++++..................###..##...++++...
~~~~~~~..............+++...........nt.....#___..#...++.....  ~~~~~~~..............+++...........n......#___..#...++.....
~~~~~~.................++@........$.......#..___#....+++...  ~~~~~~.................+$.................#..___#....+++...
~~~~~~~~~..................$..............#######.......++.  ~~~~~~~~~.................................#######.......++.
```

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

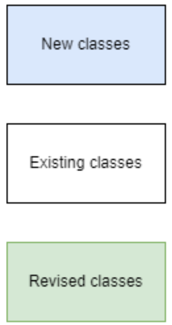
REQ1: The Overgrown Sanctuary



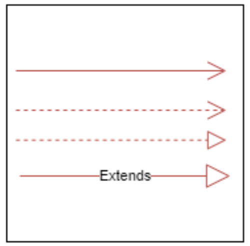
REQ2: The Blacksmith



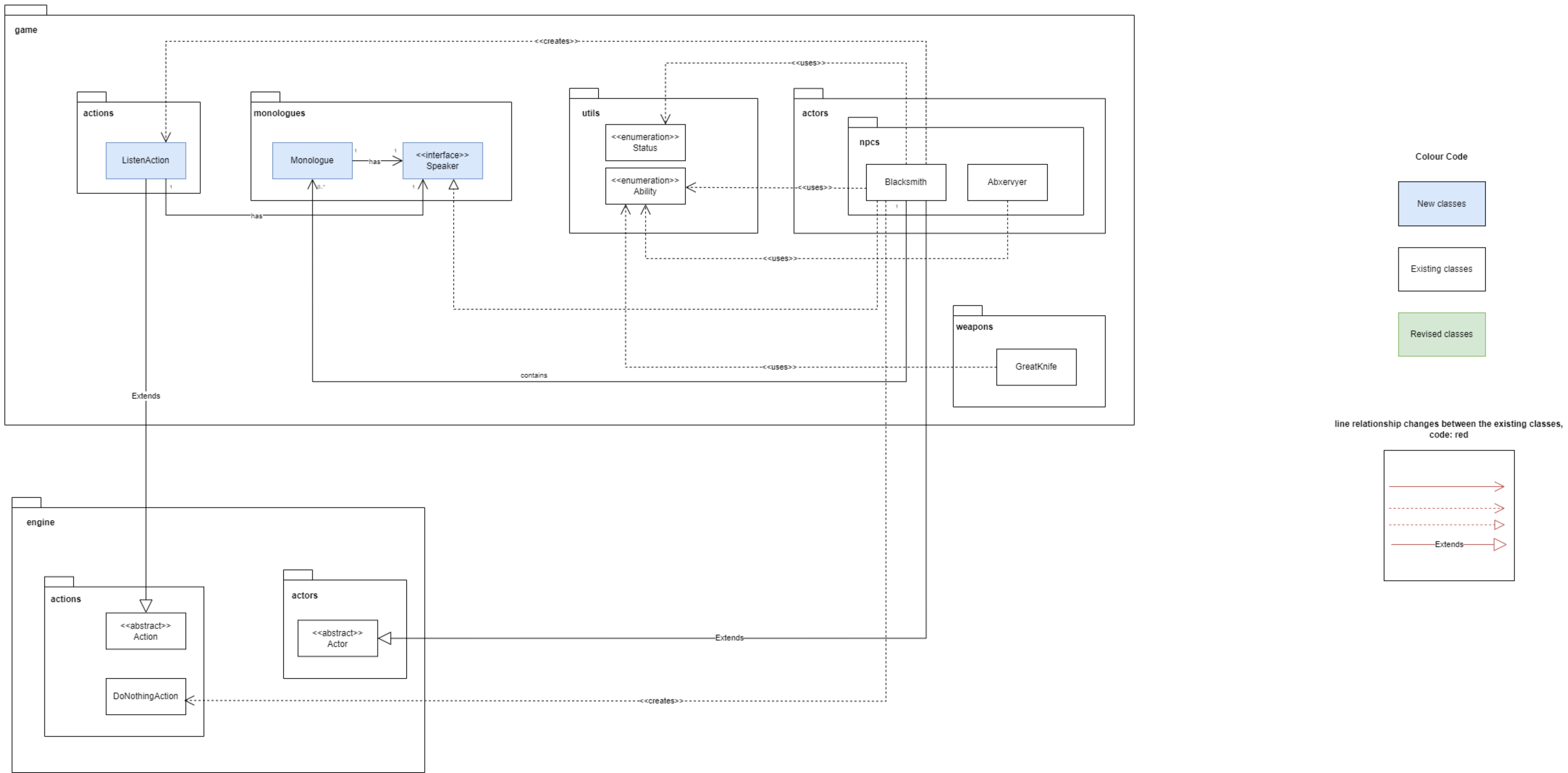
Colour Code



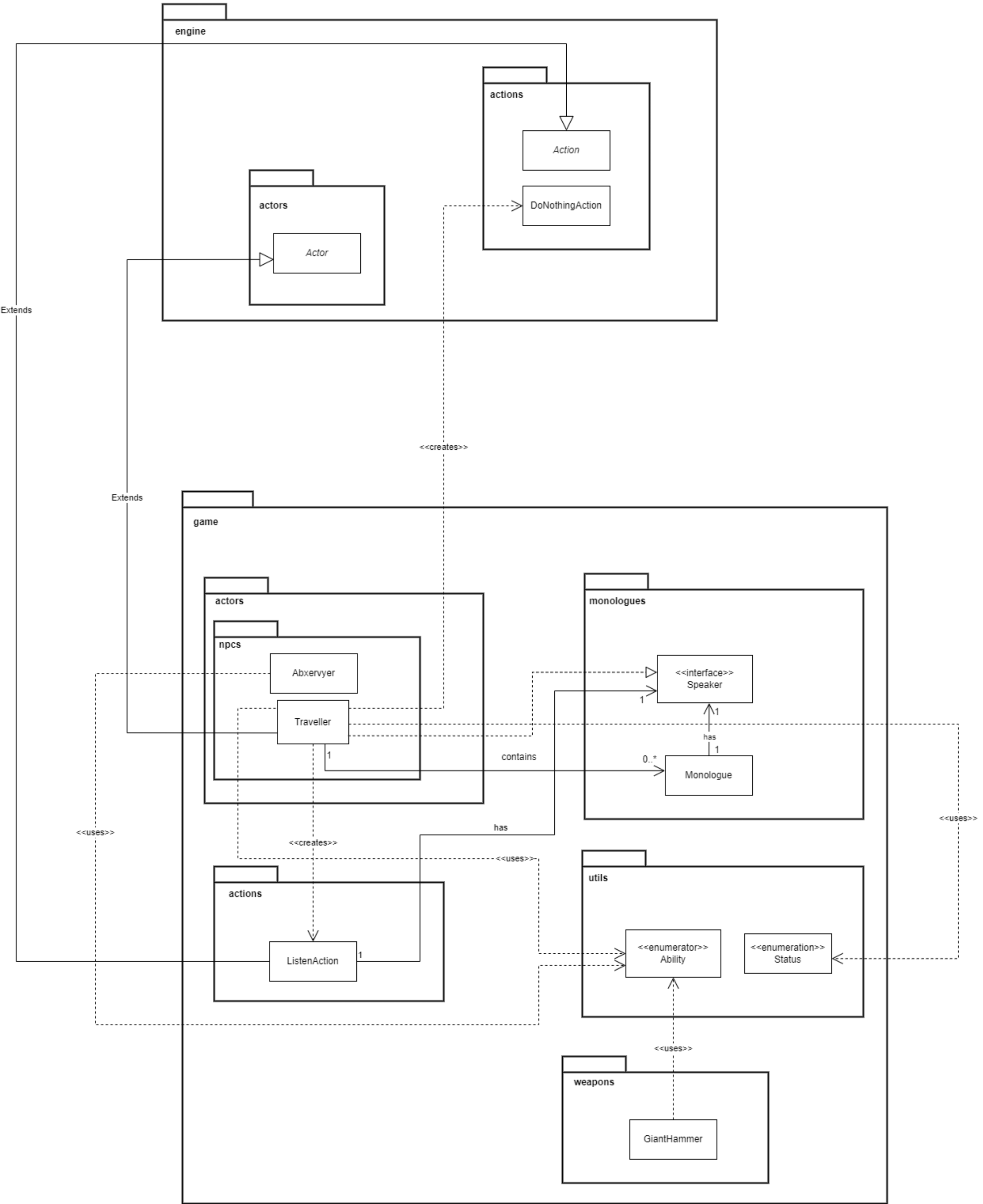
line relationship changes between the existing classes, code: red

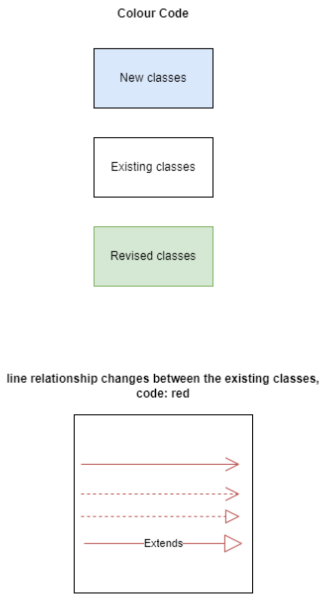


REQ 3: Conversation (Episode I)



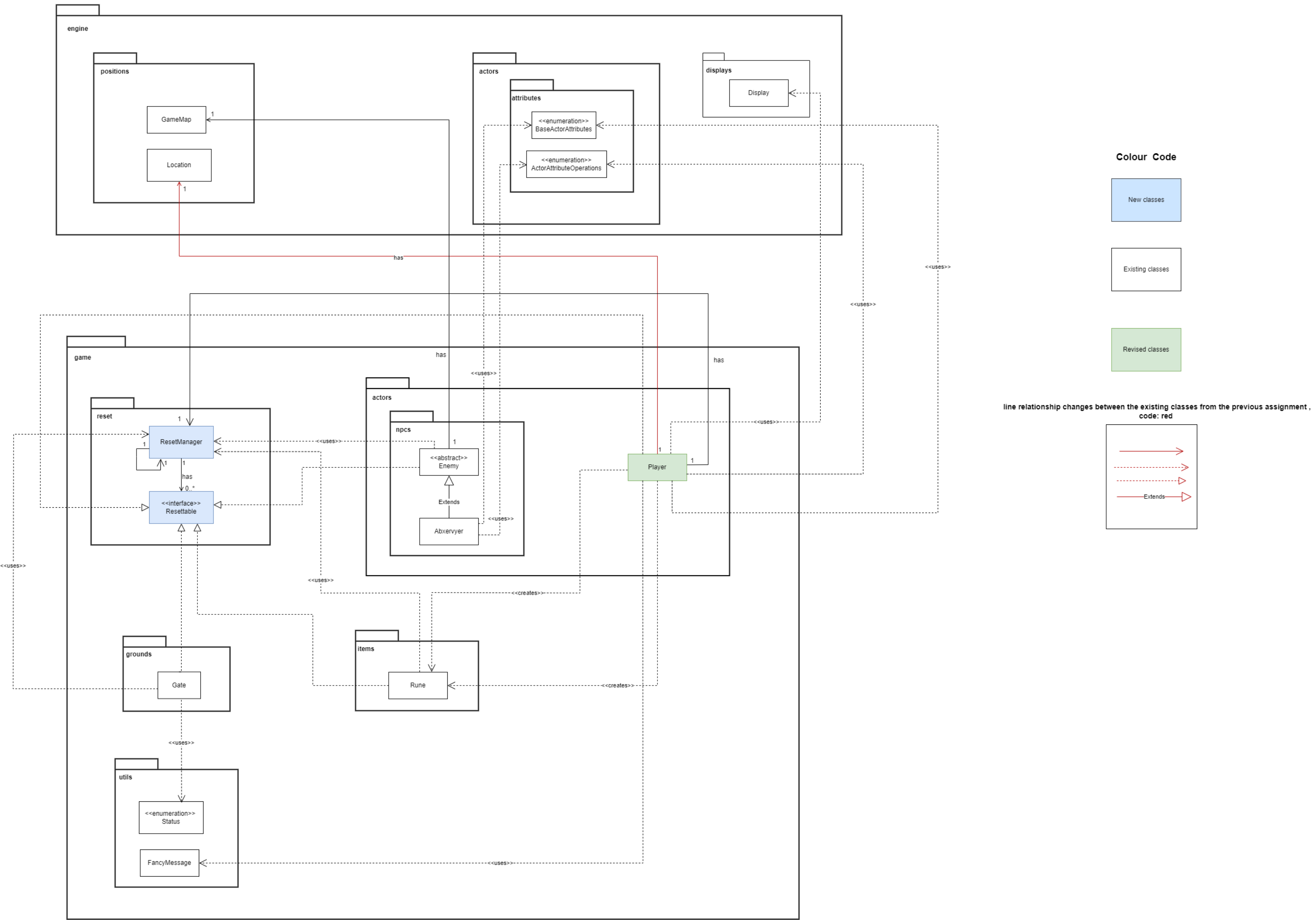
REQ 4: Conversation (Episode II)

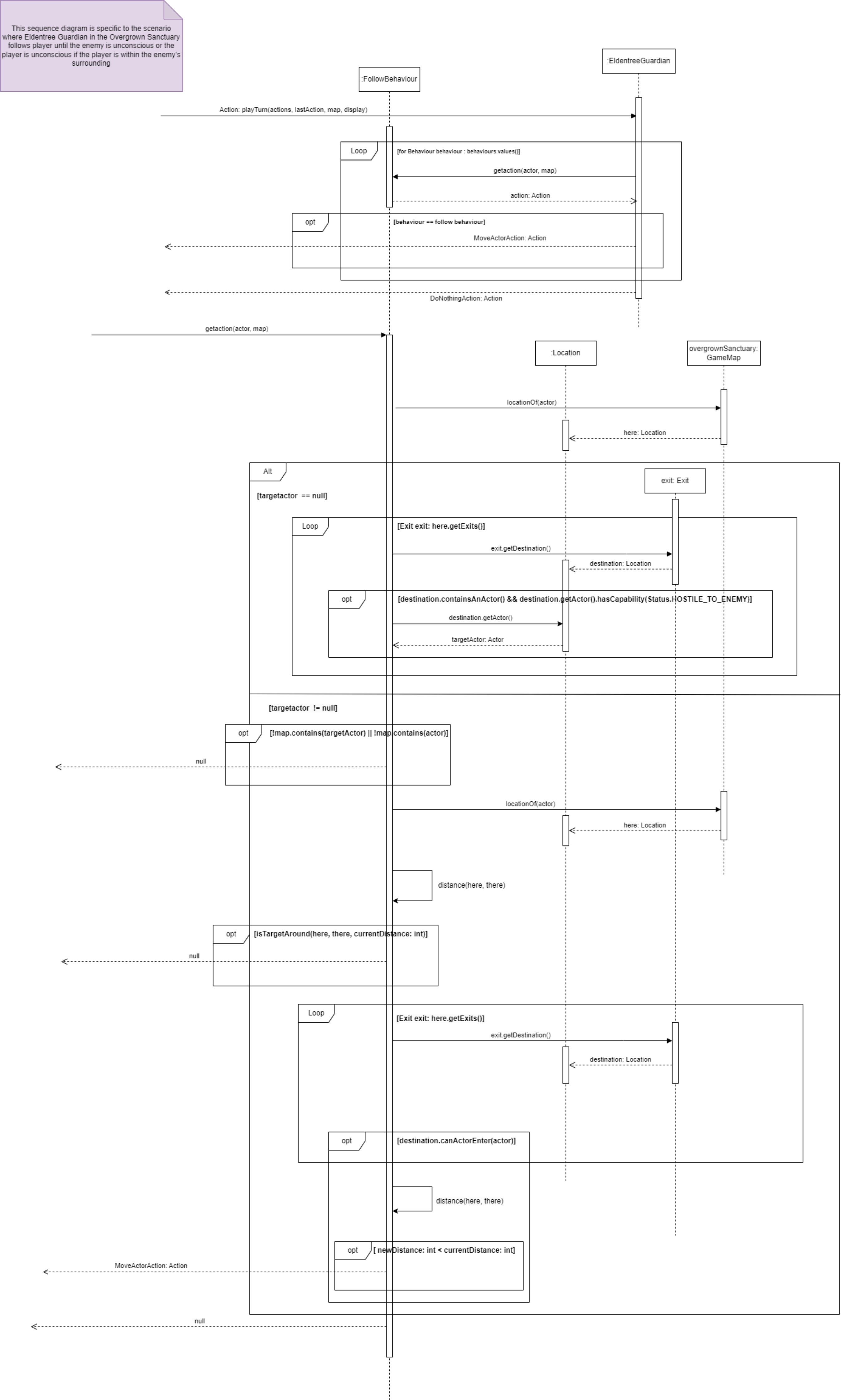




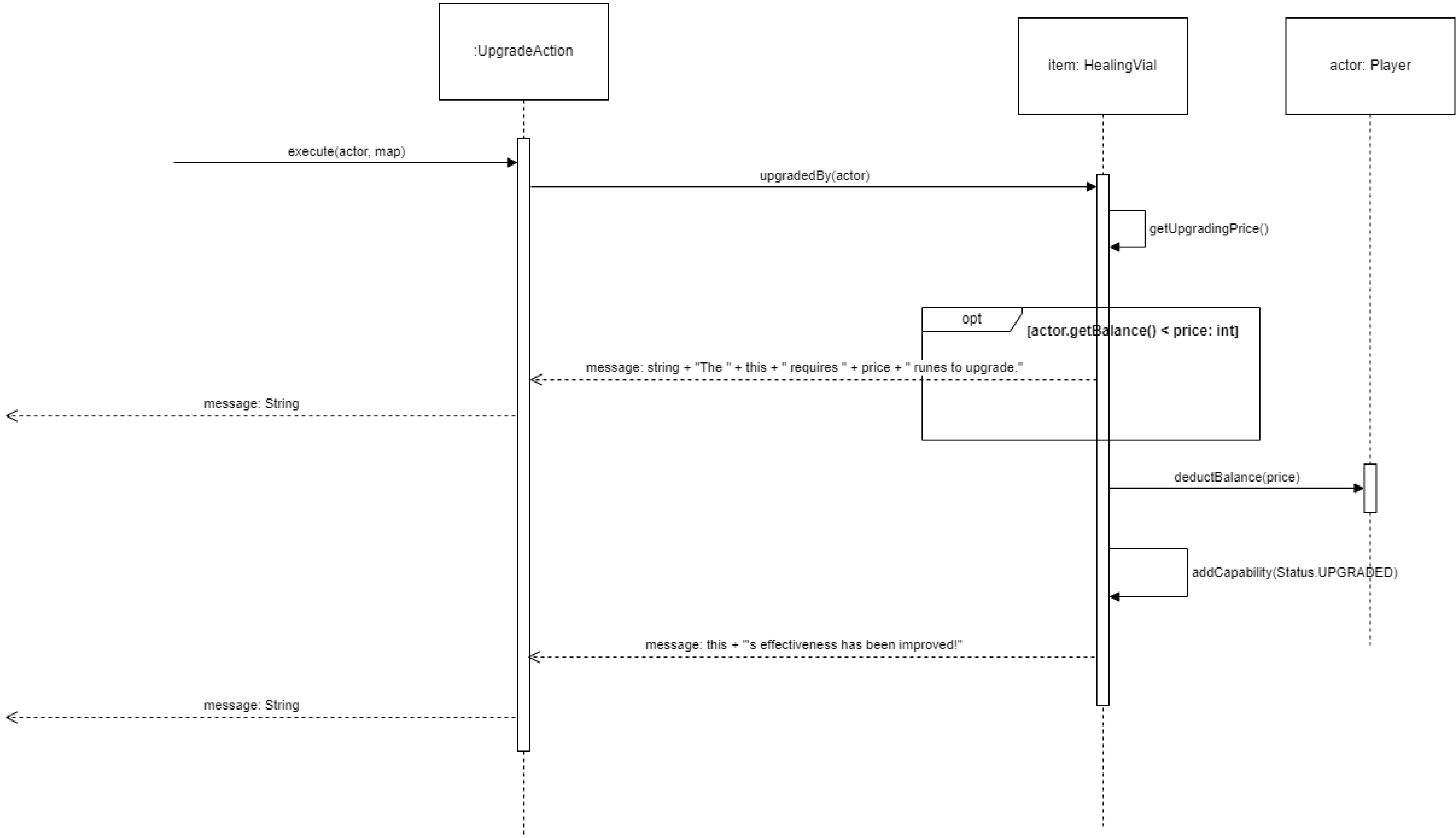
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes, code: red
Extends

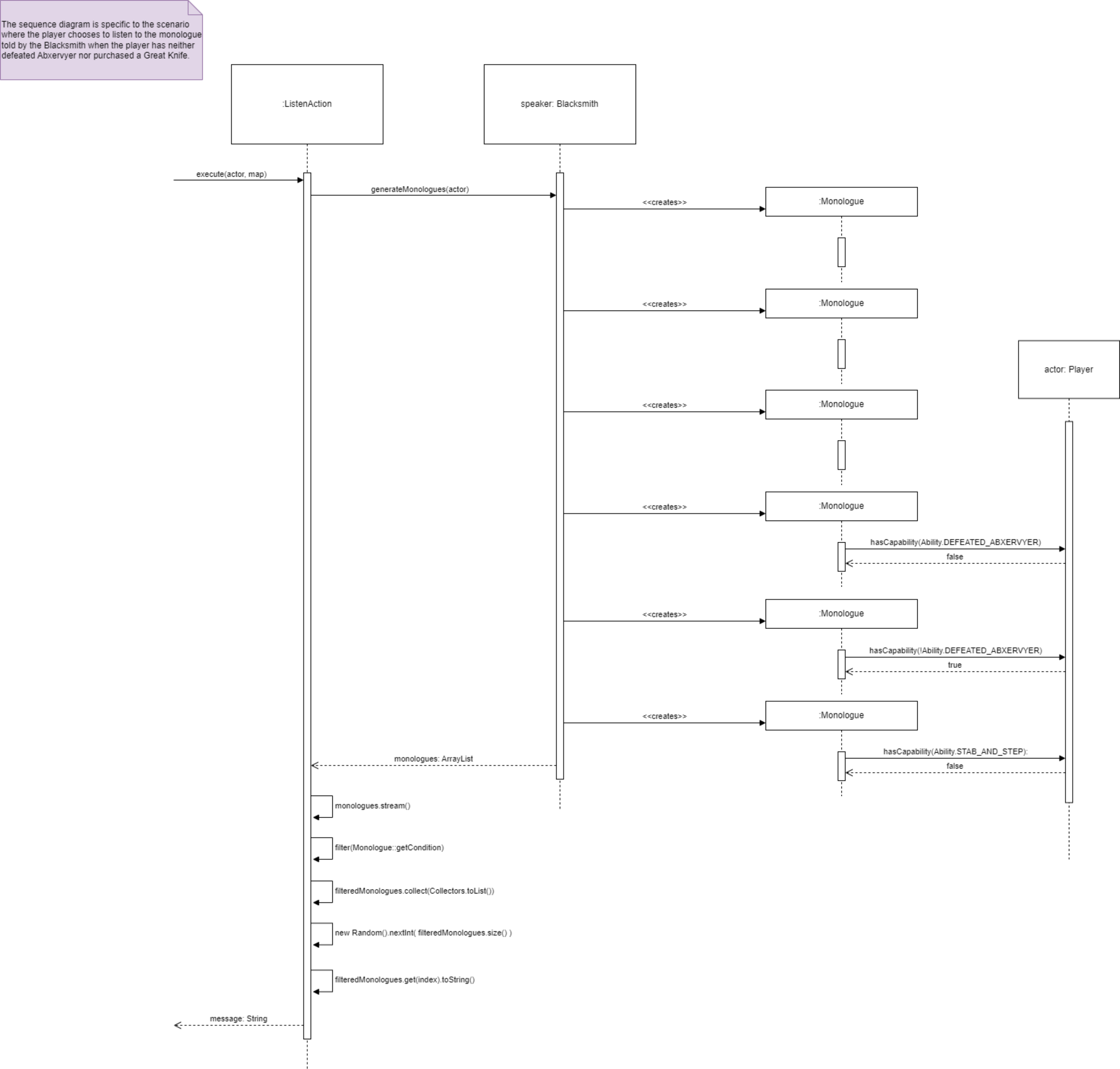
REQ5: A Dream?


engine
positions
GameMap
Location
actors
attributes
<<enumeration>> BaseActorAttributes
<<enumeration>> ActorAttributeOperations
displays
Display
game
reset
ResetManager
<<interface>> Resettable
has
0..*
actors
npcs
<<abstract>> Enemy
Extends
Abxervyer
Player
grounds
Gate
items
Rune
utils
<<enumeration>> Status
FancyMessage
has
<<uses>>
<<creates>>
1
Colour Code
New classes
Existing classes
Revised classes
line relationship changes between the existing classes from the previous assignment , code: red
Extends

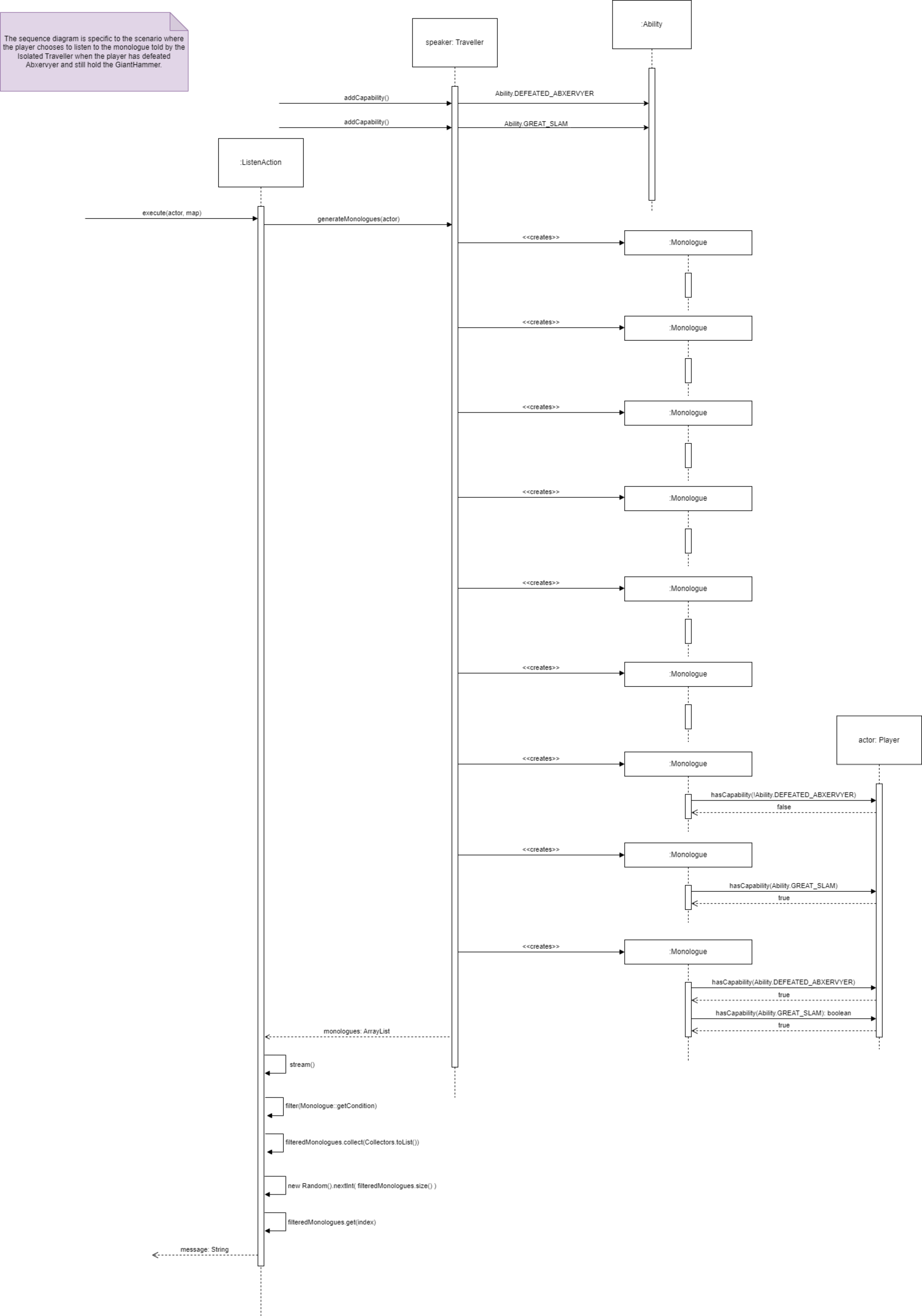

This sequence diagram is specific to the scenario where Eldentree Guardian in the Overgrown Sanctuary follows player until the enemy is unconscious or the player is unconscious if the player is within the enemy's surrounding

This sequence diagram is specific to the scenario where the Player chooses to upgrade the HealingVial from the Blacksmith in the item inventory.

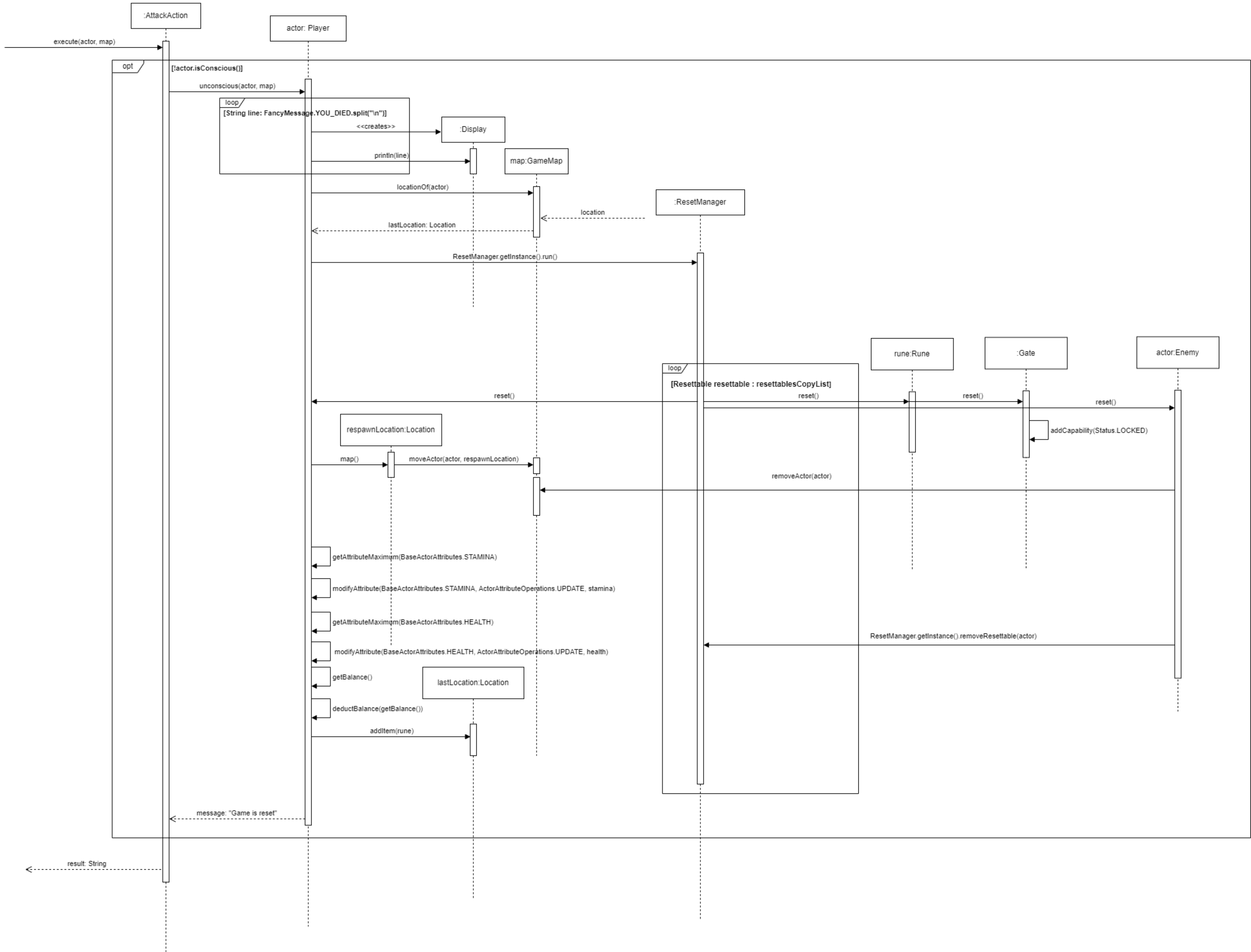

:UpgradeAction
item: HealingVial
actor: Player
execute(actor, map)
upgradedBy(actor)
getUpgradingPrice()
opt
[actor.getBalance() < price: int]
message: string + "The " + this + " requires " + price + " runes to upgrade."
message: String
deductBalance(price)
addCapability(Status.UPGRADED)
message: this + "'s effectiveness has been improved!"
message: String


The sequence diagram is specific to the scenario where the player chooses to listen to the monologue told by the Blacksmith when the player has neither defeated Abxervyer nor purchased a Great Knife.

:ListenAction
speaker: Blacksmith
:Monologue
:Monologue
:Monologue
:Monologue
:Monologue
:Monologue
actor: Player
execute(actor, map)
generateMonologues(actor)
<<creates>>
<<creates>>
<<creates>>
<<creates>>
hasCapability(Ability.DEFEATED_ABXERVYER)
false
<<creates>>
hasCapability(!Ability.DEFEATED_ABXERVYER)
true
<<creates>>
hasCapability(Ability.STAB_AND_STEP):
false
monologues: ArrayList
monologues.stream()
filter(Monologue::getCondition)
filteredMonologues.collect(Collectors.toList())
new Random().nextInt(filteredMonologues.size())
filteredMonologues.get(index).toString()
message: String


The sequence diagram is specific to the scenario where the player chooses to listen to the monologue told by the Isolated Traveller when the player has defeated Abxervyer and still hold the GiantHammer.

The sequence diagram is specific to the scenario where the Player is unconscious due to the other actor then causes the game to reset.


:AttackAction
actor: Player
execute(actor, map)
opt
[!actor.isConscious()]
unconscious(actor, map)
loop
[String line: FancyMessage.YOU_DIED.split("\n")]
<<creates>>
:Display
println(line)
map:GameMap
locationOf(actor)
:ResetManager
location
lastLocation: Location
ResetManager.getInstance().run()
rune:Rune
:Gate
actor:Enemy
loop
[Resettable resettable : resettablesCopyList]
reset()
reset()
reset()
reset()
respawnLocation:Location
addCapability(Status.LOCKED)
map()
moveActor(actor, respawnLocation)
removeActor(actor)
getAttributeMaximum(BaseActorAttributes.STAMINA)
modifyAttribute(BaseActorAttributes.STAMINA, ActorAttributeOperations.UPDATE, stamina)
getAttributeMaximum(BaseActorAttributes.HEALTH)
ResetManager.getInstance().removeResettable(actor)
modifyAttribute(BaseActorAttributes.HEALTH, ActorAttributeOperations.UPDATE, health)
getBalance()
lastLocation:Location
deductBalance(getBalance())
addItem(rune)
message: "Game is reset"
result: String