Design Rationale

By: MA_AppliedSession1_Group3

REQ4: Conversation (Episode II)

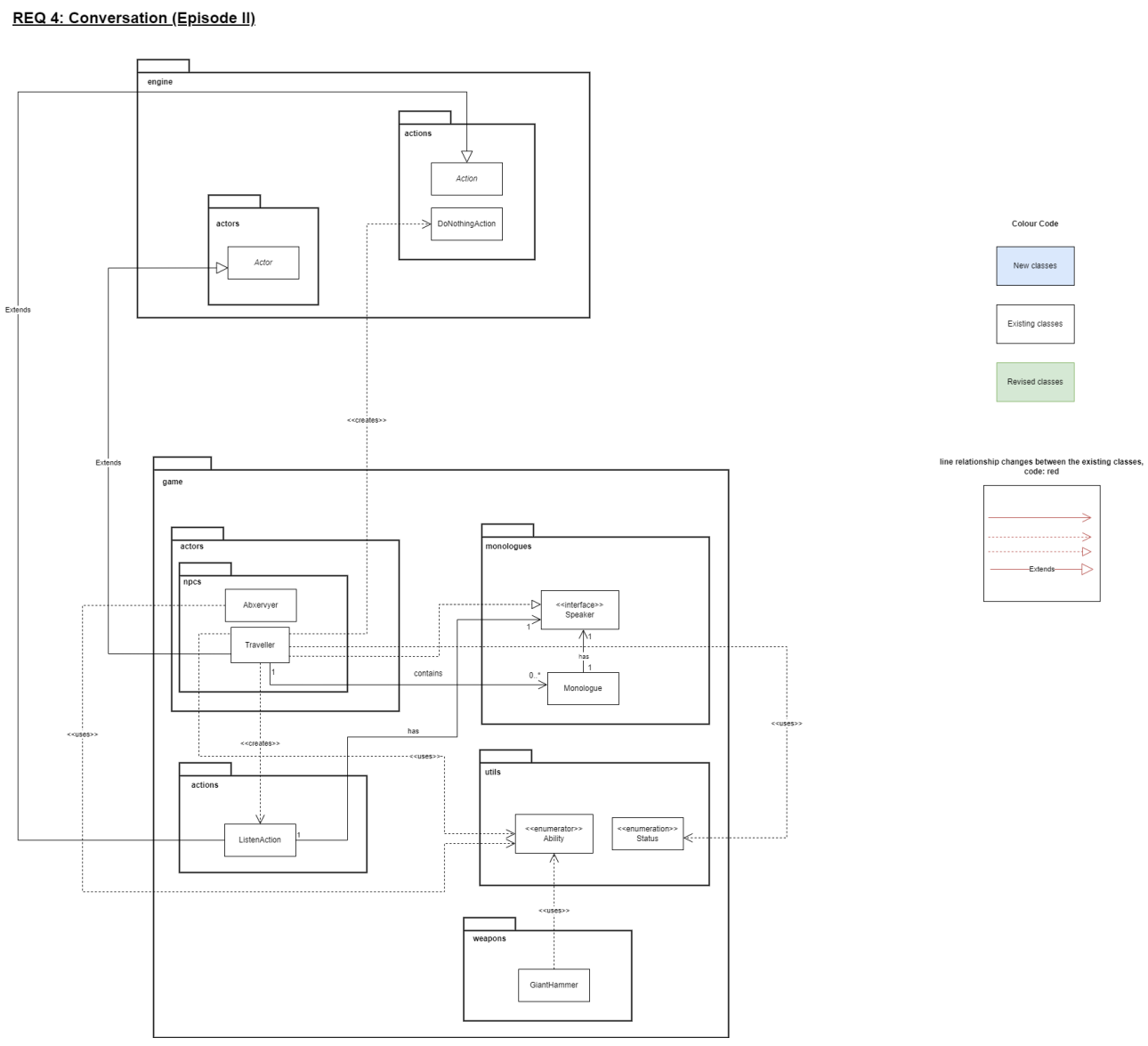

Figure 1: UML class diagram of REQ4: Conversation (Episode II)

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)**.