Design Rationale

By: MA_AppliedSession1_Group3

REQ3: Conversation (Episode I)

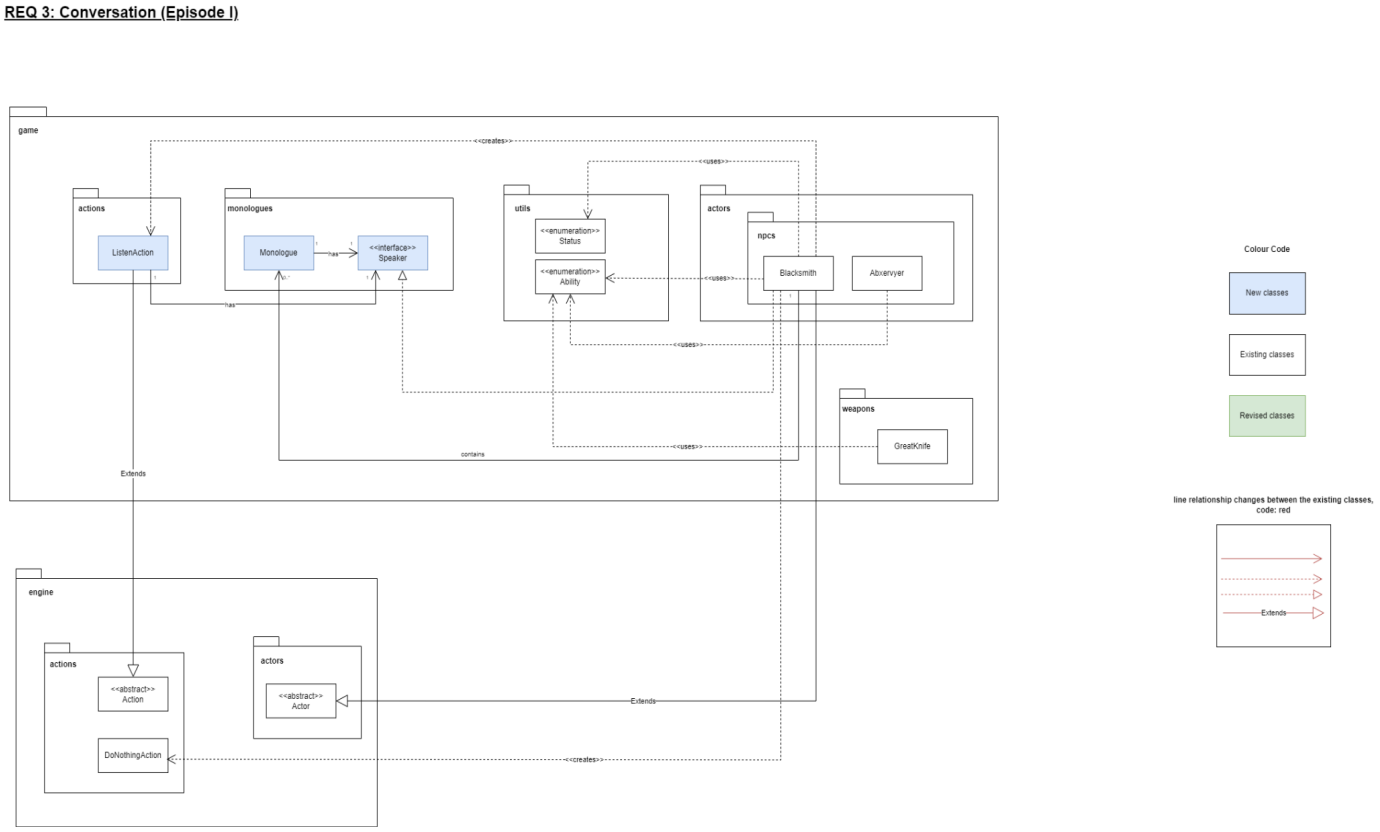

Figure 1: UML class diagram of REQ3 - Conversation (Episode I)

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).