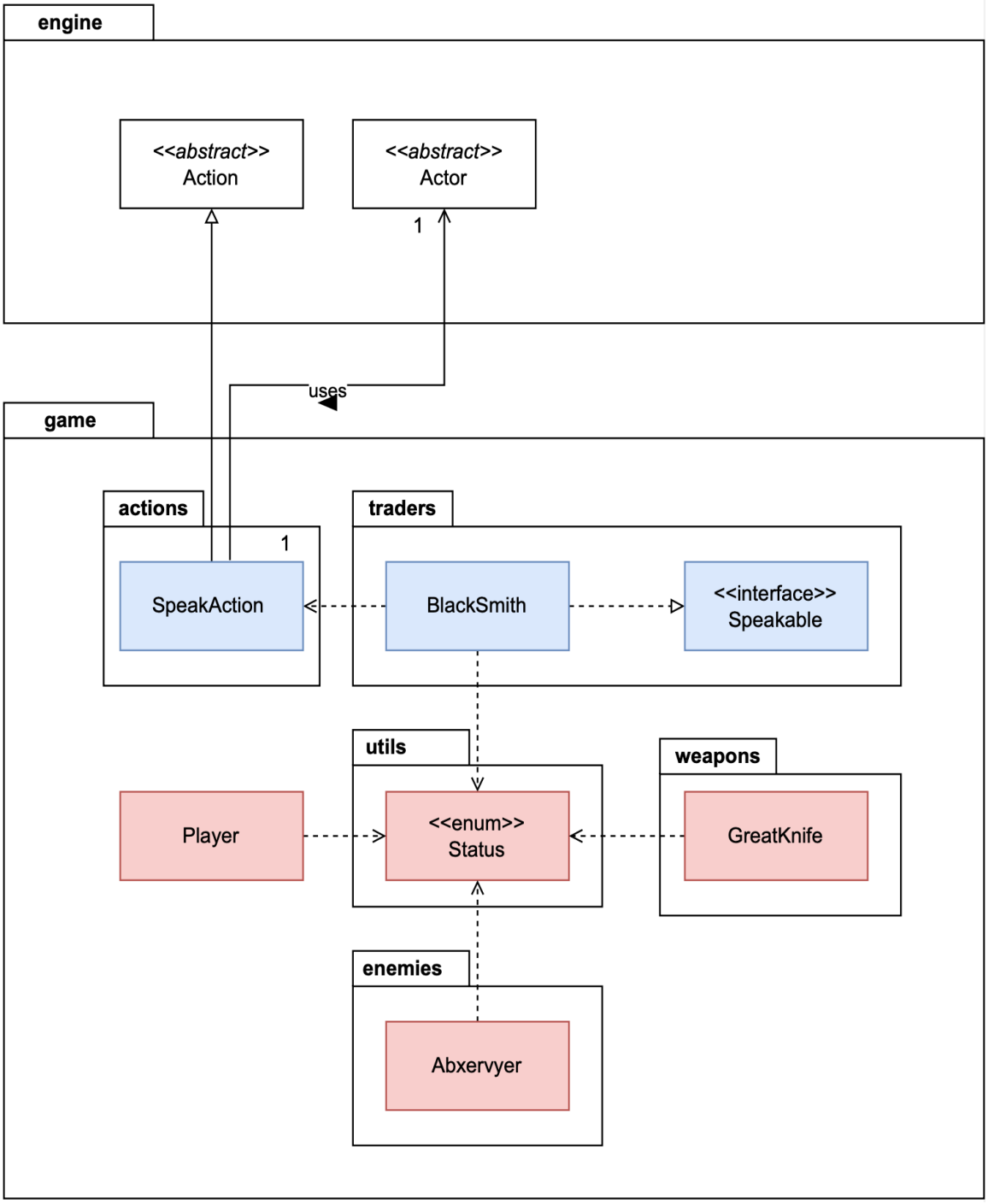
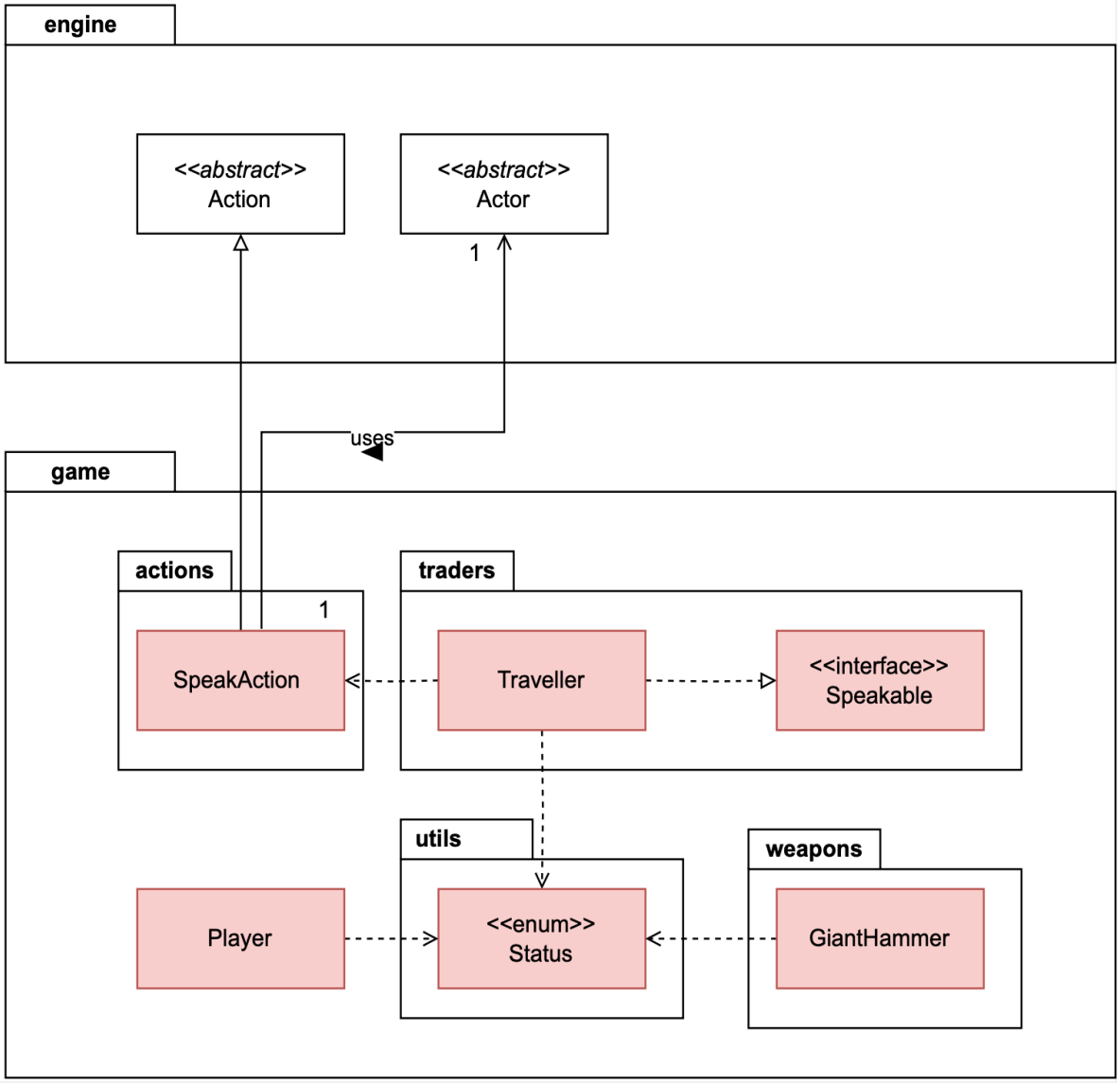
Design Rationale (REQ3, 4)

UML Diagram

REQ 3: Conversation (Episode I)



REQ 4: Conversation (Episode II)



Classes Added

- Class SpeakAction extends Action
- Interface Speakable

* Since both Requirement 3 and 4 are very similar, we will explain the rationale behind these two requirements together here.

Design Rationale

Interface Speakable

The **Speakable** interface represents specific actors in the game that are capable of speaking dialogues, such as the **Blacksmith** (for Requirement 3) and the **Traveller** (for Requirement 4). This interface contains two methods: **getDialogue**() and **generateDialogue**(). Both the **Blacksmith** and **Traveller** classes implement this interface.

The **getDialogue**() method is designed to return a random dialogue from the **dialogue** ArrayList, which is maintained within the **Blacksmith** and **Traveller** classes. On the other hand, the **generateDialogue**() method is responsible for populating the **dialogue** ArrayList with default dialogues.

In the **allowableActions**() method of both the **Blacksmith** and **Traveller** classes, the dialogue list is reset to a new, empty ArrayList. Subsequently, dialogues are added to the list based on the current game conditions (e.g., whether the player has defeated a boss, is holding a great knife, etc.). Finally, the **generateDialogue**() method is called to add the default dialogues to the list, and a random dialogue is then returned by the **getDialogue**() method.

Principles Followed

Single Responsibility Principle (SRP)

- The **Speakable** interface adheres to the Single Responsibility Principle as it has one clear responsibility: to define behaviours for actors that can speak in the game. By doing so, the interface ensures that each class implementing it needs only to focus on the specifics of generating and retrieving dialogues.

Interface Segregation Principle (ISP)

- The principle is maintained as the **Speakable** interface provides specific methods related to speaking functionality, ensuring that implementing classes are not forced to depend on methods they do not use.

High Cohesion

- The **Speakable** interface promotes high cohesion by grouping related functionalities (speaking dialogues) together, ensuring that all methods within the interface are aligned with the core responsibility of speaking.

Low Coupling

- The interface promotes low coupling as it allows for flexible implementations. Classes that implement the **Speakable** interface are not tightly bound to specific implementations, enhancing modularity and ease of maintenance.

Cons of the Design

- Since each class has to manage the **dialogue** ArrayList and conditions independently, there is a risk of code duplication, particularly in the handling of default dialogues and conditional checks. However, based on the requirement provided, the default dialogues for both the **Blacksmith** and **Traveller** classes are different, and the condition check for different dialogues are also different. Hence this should not be a problem for this design.

Class SpeakAction

The **SpeakAction** class represents the action of delivering a dialogue. In requirements 3 and 4, both the **Blacksmith** and the **Travellers** have the capability to communicate with the player, necessitating the inclusion of the **SpeakAction** within their respective **allowableActions**() methods.

The **SpeakAction** class requires two parameters for instantiation: **spokenSentence** and **target**. The **spokenSentence** represents the dialogue to be delivered, while the **target** denotes the actor who will be conveying the message to the player, which could be either the **Blacksmith** or the **Travellers**.

To facilitate the determination of which specific dialogue should be returned based on game conditions, we have employed several enums, such as **Status.DEFEATED_BOSS, Status.HOLDING_GREAT_KNIFE, Status.HOLDING_GIANT_HAMMER, Status.IS_GREAT_KNIFE, and Status.IS_GIANT_HAMMER**. This approach simplifies the code and ensures a more straightforward handling of various game situations.

For instance, upon the player's defeat of the boss, the **DEFEATED_BOSS** status is assigned, similar to how the **HOLDING_GIANT_HAMMER** and **HOLDING_GREAT_KNIFE** statuses are applied to the player when they possess the corresponding items. On the other hand, the statuses **IS_GREAT_KNIFE** and **IS_GIANT_HAMMER** are attributed to the specific weapons themselves. Consequently, if a player is holding a weapon with the **IS_GREAT_KNIFE** status, the **HOLDING_GREAT_KNIFE** status is also assigned to the player. This principle is consistently applied across different statuses and game conditions.

Design Principles Followed

Don’t Repeat Yourself (DRY)

- The SpeakAction class adheres to the DRY principle as it encapsulates the dialogue delivery behaviour in a single place, preventing the need for code repetition across different parts of the program.

Single Responsibility Principle (SRP)

- The **SpeakAction** class has a well-defined and singular responsibilty – managing the action of an actor speaking a sentence. It encapsulates all the detaills necessary for this action, such as the spoken sentence and the target actor.
- This adherence promotes a clean design, as changes to how speaking actions are handled will only necessitate modifications within this class.

High Cohesion

- The class is highly cohesive, as all of its parts are related to the single responsibility of representing a speaking action.

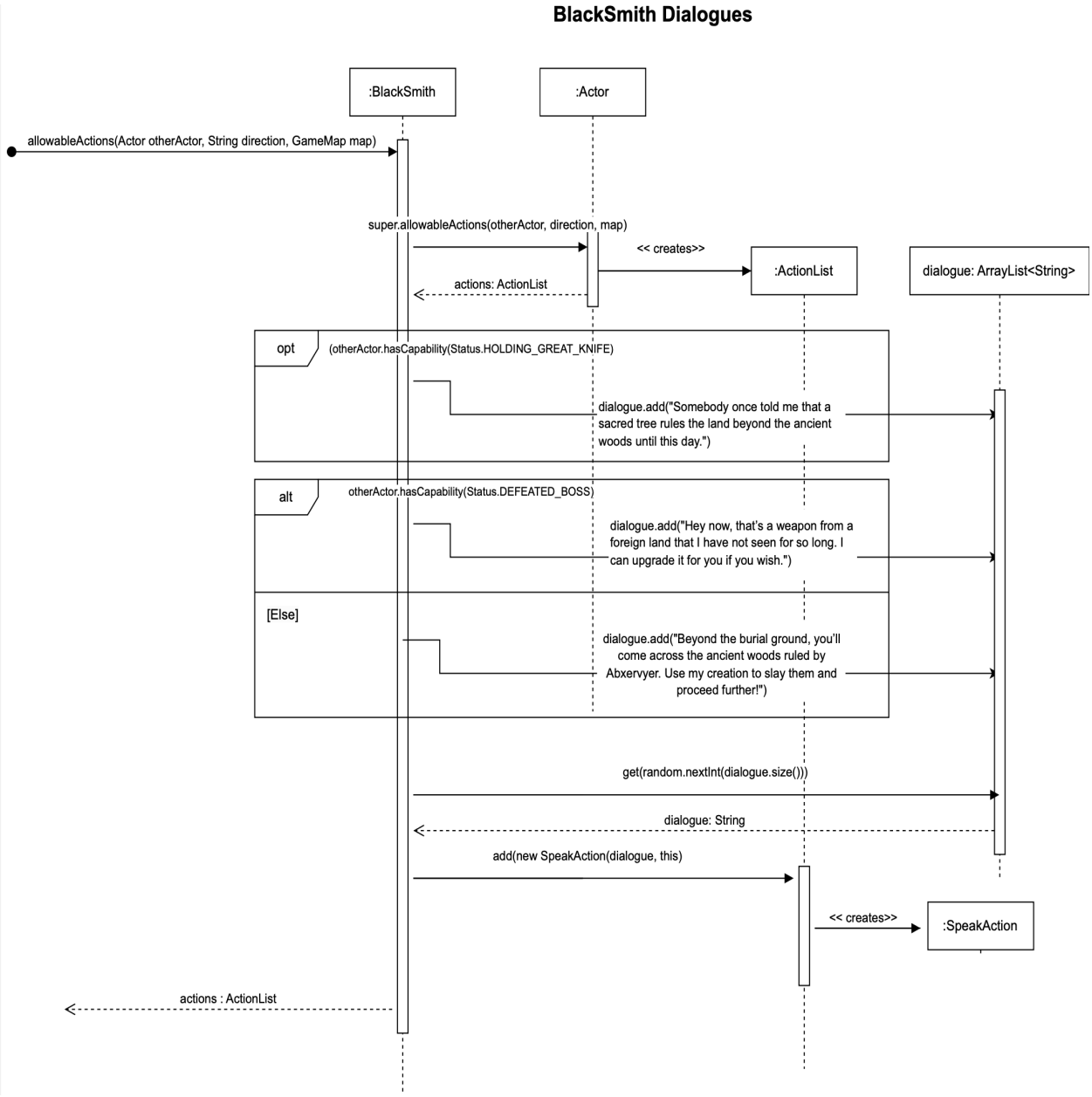
Low Coupling

- The class is loosely coupled with the rest of the system. It depends on the **Actor** and **Action** classes, but these are abstract or base classes, which means the **SpeakAction** class is not tightly bound to specific implementations.

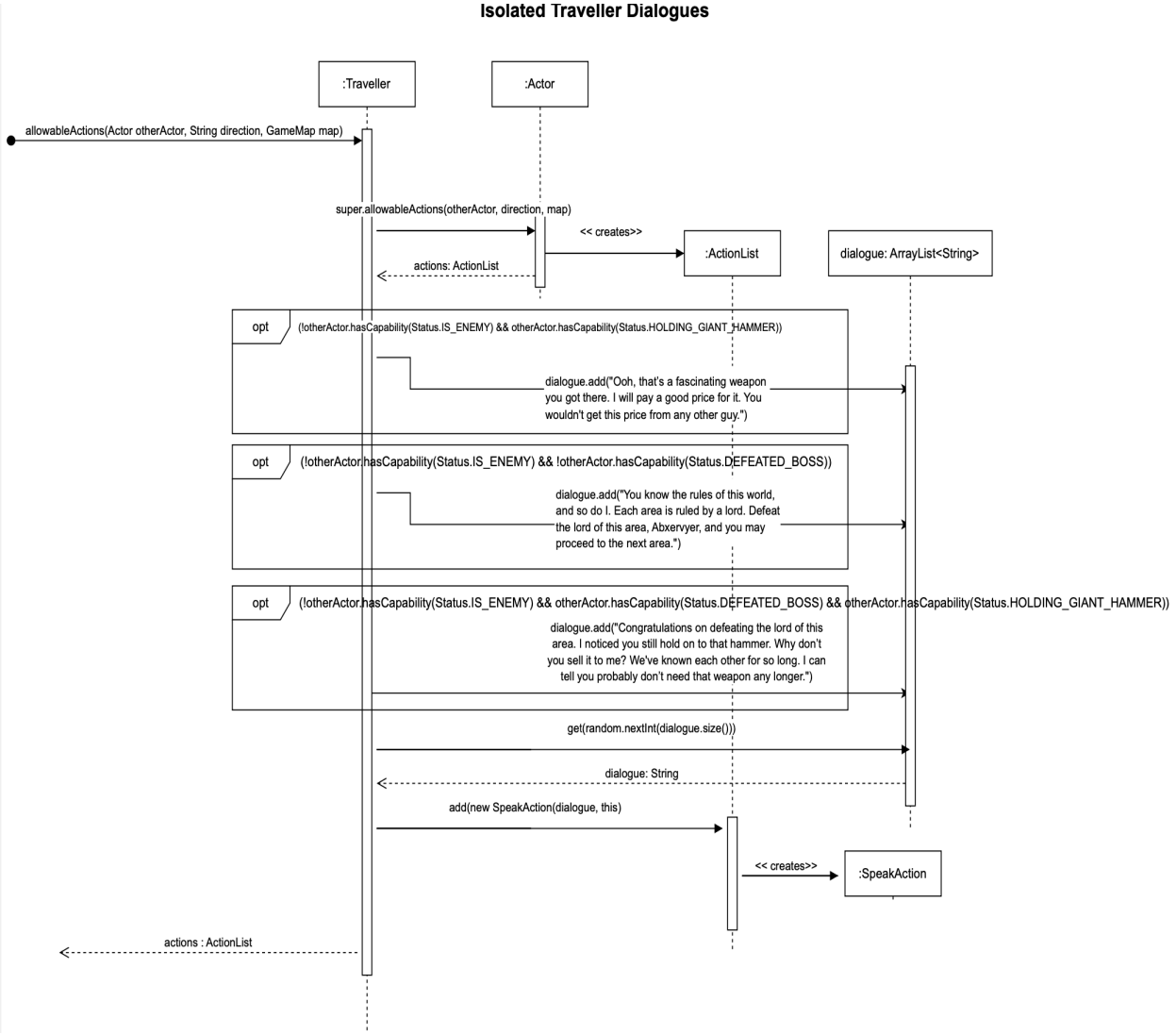
Cons of the Design

- The addition of the new attributes brings more complexity to the system and goes against the principle of ReD (reduce dependency). Increase in the number of player attributes will not only reduce the maintainability of the system, but also will violate the principle of ReD (reduce dependency) as the number of dependencies on the attributes increase in the future.
- Enum typically reside in a global namespace, and if many enums are defined, this can lead to namespace pollution. This might result in naming conflicts or difficulties in finding the appropriate enum values to use.
- Using enum is just same as using the instanceof(). It can be considered bad practise as it violates Open-Closed Principle. Using this often leads to code structures that have to be modified whenever a new subclass or type is added, where we might need to add additional checks and corresponding behaviour.

Sequence Diagram for Requirement 3



Sequence Diagram for Requirement 4



Comparison to the Assignment 1 Sample Solution

Addressing the Questions:

1. **How many classes would you need to create for this requirement if you used the Assignment 1 sample solution?**

   No changes were made in the implementation in comparison to the sample Assignment 1 solution other than the addition of the Speakable interface as the Blacksmith class extends the abstract classes (Actor, Action respectively) readily made in the engine package.

2. **Do they contain some code duplications?**

   No duplications were found in our implementation. Although there are multiple if or if-else statements within allowableActions of each class that implements the Speakable interface, we have concluded that they are necessary to validate conditions for adding appropriate dialogues.

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

   A possible alternative design could be creating an additional class DialogueManager to store/add all dialogues that are spoken by speakable objects. This would significantly reduce the complexity of the code in the future when more dialogues are to be added and handled by each class. However, this would add more complexity to the system as the inevitable dependencies between the Manager class and speakable classes would violate ReD. Furthermore, this design fails to encapsulate the class-specific data (in this case, the strings of dialogue) as any speakable class can access each other's dialogue.

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

   We opted for the design involving a Speakable interface to be implemented by any class that is capable of implementing dialogues, as demonstrated in the provided code. The design choice promotes code readability, maintainability, and ease of extension, aligning with the principle of SOLID principles.