Design Rationale for Requirement 3

ListenAction

ListenAction class was extended from the abstract Action class and had a one-to-one association with the TalkableEntity interface. This class was used for any entities that implemented the TalkableEntity interface to be able to say their monologues by simply calling the talked method. Thus, it adhered to the Open-closed and Liskov Substitution principles by abstracting out the direct relationship with any speakable entities and still being able to swap the concrete classes with the interface without any difficulties.

Getting the monologues

Listening to monologues from different traders can be achieved using two different approaches. These approaches are discussed in the following section, highlighting their pros and cons to determine the best approach.

Method 1. TalkableEntity and TalkingMaterial interfaces

The implementation of the conversations was based on the Observer design pattern, in which TalkableEntity was the subject interface, and TalkableMaterial was the observer interface. All entities that could speak, such as the Blacksmith, implemented the former, allowing them to deliver their monologues to the player under certain conditions. The entities that play the roles of the materials for the monologue's requirements implemented the TalkingMaterial and informed the TalkableEntity of the phrases they could say. There was a centralised array list of TalkingMaterial interfaces in each talkable concrete class to control the flow of information, which would be used to determine which phrases could be spoken.

Two major drawbacks of this implementation were all talkable entities would become god classes, in which they had to monitor the flow of information issued by the TalkingMaterials. This would happen by adding and removing them from the controlling array and then directly calling all the materials to get the phrases. To be more specific, TalkingMaterial had a getPhrase method, which returned either the terms associated with that material or nothing; thus, this was a violation of Single Responsibility, Open-closed, and Interface Segregation principles. The TalkingMaterial should not have been responsible for returning the phrases that the talkable entity could say but should have just informed a piece of information that might be used for the talking conditions to be met. Finally, it would need to be more extensive to directly return the phrases that only the Blacksmith could say, not other entities that might be added.

A significant improvement that resolved the abovementioned drawbacks would be introduced below.

Method 2. Monologue and Condition

To resolve the above critical disadvantages, an application of the Mediator design pattern was used, in which the informational materials for the conversations were notified through the third-party concrete implementations of the Condition interface. Going from the outermost layer, all of the Monologue options, with their triggered list of conditions, were set in the Application construction as a List, and they were passed into the respective concrete TalkableEntity as a parameter for the constructor. Then, once the talked method was triggered in the ListenAction's execute method, it looped through all of the Monologue and checked for individual canBeSpoken methods. Inside this canBeSpoken method, it continued to loop through the list of conditions, returning a boolean with whether the conditions were met. Back to the talked method, if all conditions were met, the getPhrase method was called to get the phrases stored in the concrete classes and pushed to the availablePhrases list. Lastly, a randomiser was used to randomly select a phrase in the availablePhrases list and return it.

This approach ticked all of the SOLID and DRY principles by reducing the responsibility of directly gathering information and getting the appropriate phrase of the Blacksmith class. With the old implementation, due to the phrases being returned as a string, it left no room for extensibility; however, with an array of phrases, it gives the option for multiple strings to be used or concatenated in the future. Finally, this implementation chains up the conditions so that if a phrase needs many conditions to be met before being said, it would still be sufficient to achieve it without changing the whole implementation.

Finally, the creation of the monologue was used for sequential diagram demonstration due to its high complexity of needing to interact with the condition concrete classes and usages of multiple list creation.