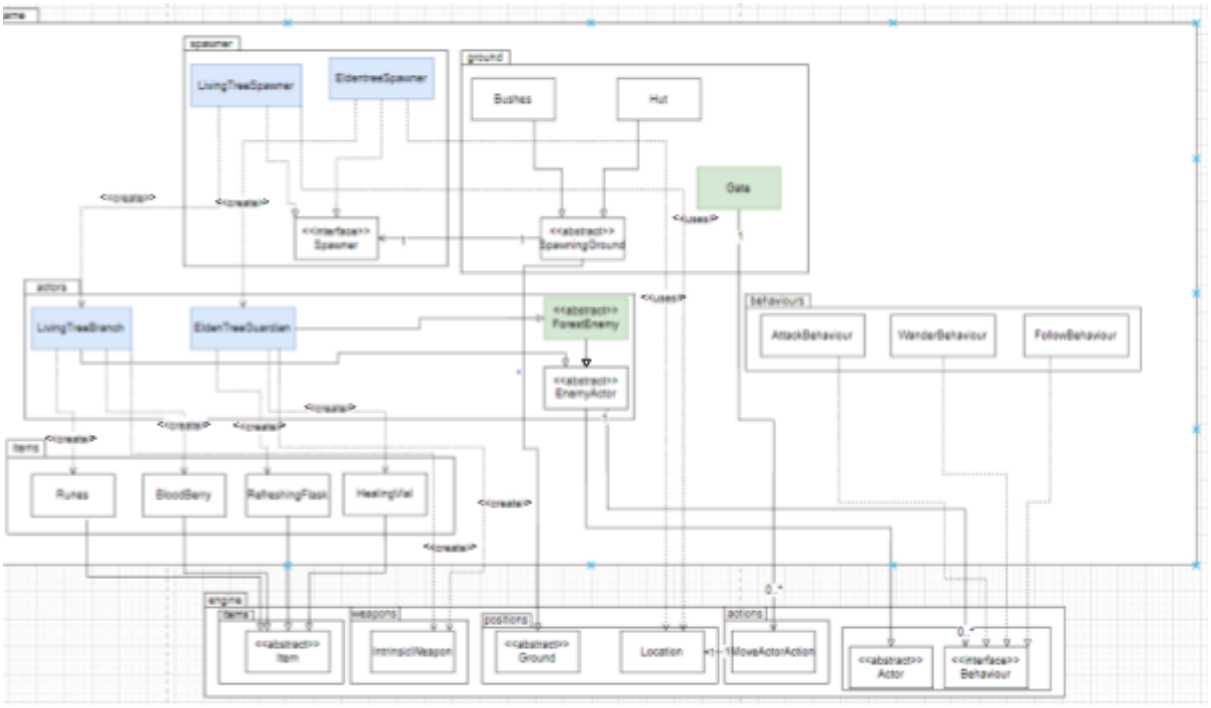
Req 1 UML Diagram

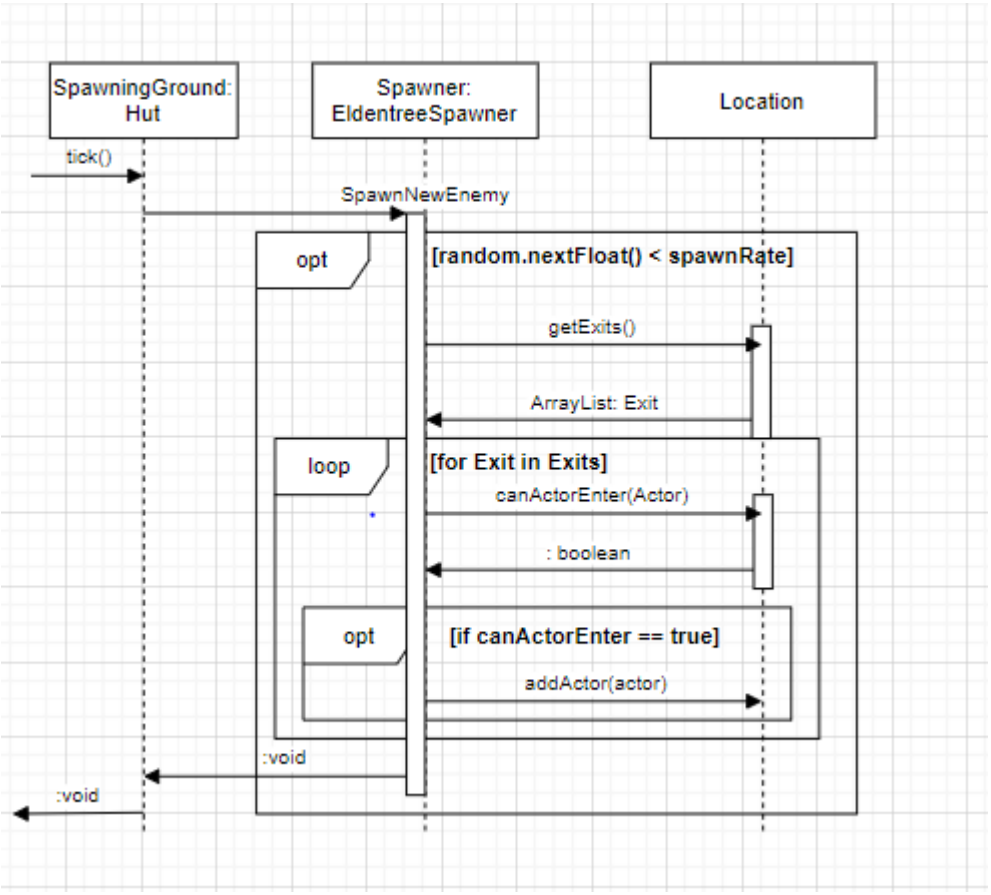


Green - refactor from previous implementation

Blue - new implementation

Req 1 Sequence Diagram

This sequence diagram depicts the EldentreeSpawner spawning a new enemy.



Req 1 Design Rationale

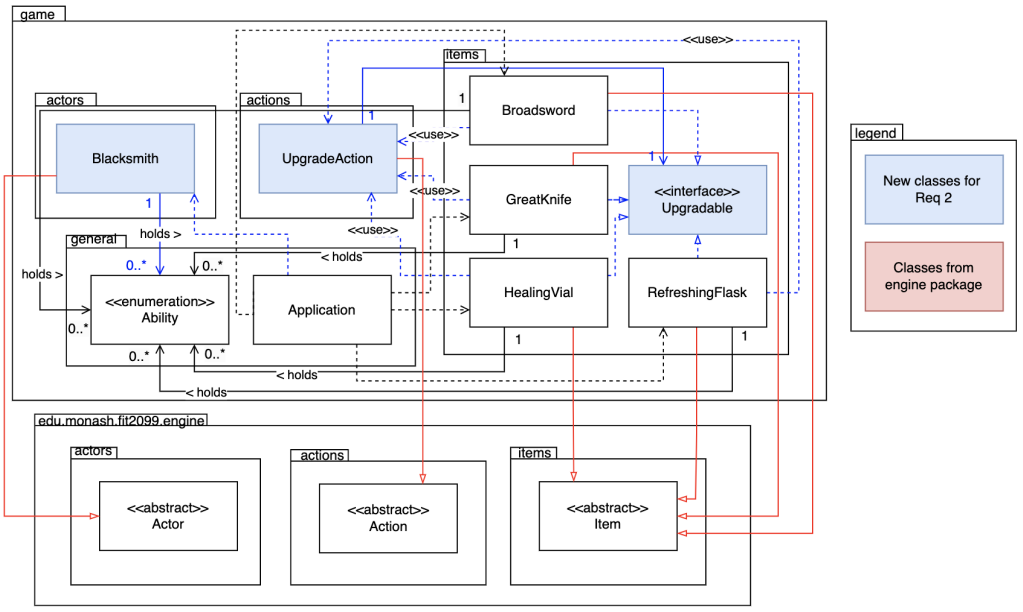
For requirement 1 we implemented 5 new classes (LivingTreeBranch, EldentreeGuardian, LivingTreeSpawner, EldentreeSpawner and ForestEnemy) and modified the Gate class. The gate class was changed in line with the specification to store an ArrayList of MoveActorActions, storing the actions in an ArrayList aligns with the Open and Closed Principle (OCP) as it provides a flexible mechanism to extend the functionality of the gate.

The ForestEnemy abstract class extends the EnemyActor abstract class and contains the contract for enemy actors that follow the player. This change to the implementation from the previous assignment adheres to the DRY as enemies that follow the player contain duplicate code it also adheres to the Open and Closed Principle (OCP) as new enemies that follow the player can extend this class and it does not require modification of an old class, every time you implement a new enemy with the FollowBehaviour.

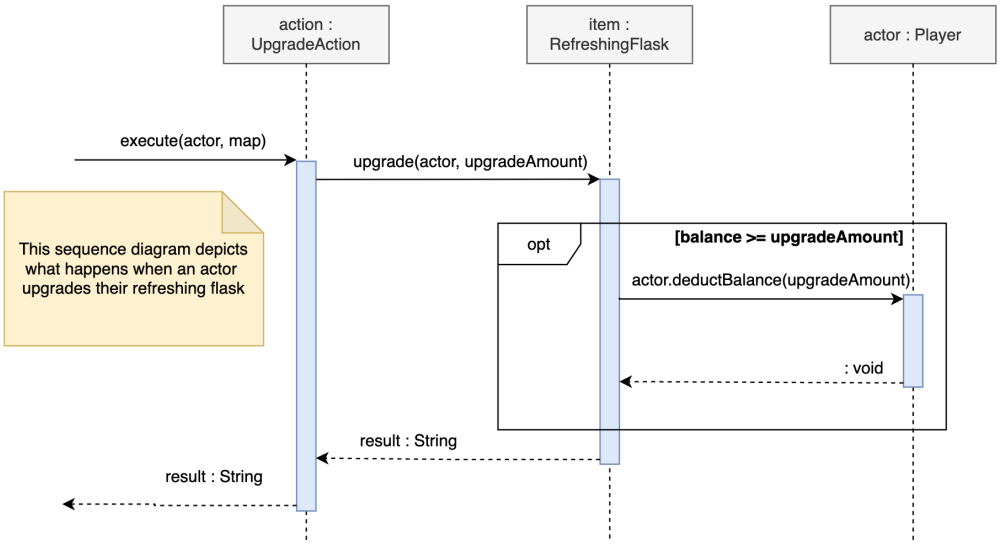
We created new classes for the LivingTreeSpawner and EldentreeSpawner which implement the Spawner interface, as well as the LivingTreeBranch and EldentreeGuardian classes that extend the EnemyActor and ForestEnemy abstract class respectively. This adheres to the Single Responsibility Principle (SRP) as each spawner class contains the contract for the spawning of each enemy class and each enemy class contains the responsibilities of the new EnemyActor.

Potential downsides of this implementation include the addition of the new ForestEnemy abstract class which could be seen as an excess abstract class given it only holds a few lines of new code and is a 3rd generation extension abstract class, which could make the code messy and confusing. However it aligns well with the principles of DRY and OCP as previously stated.

Req 2 UML Diagram



Req 2 Sequence Diagram



Req 2 Design Rationale

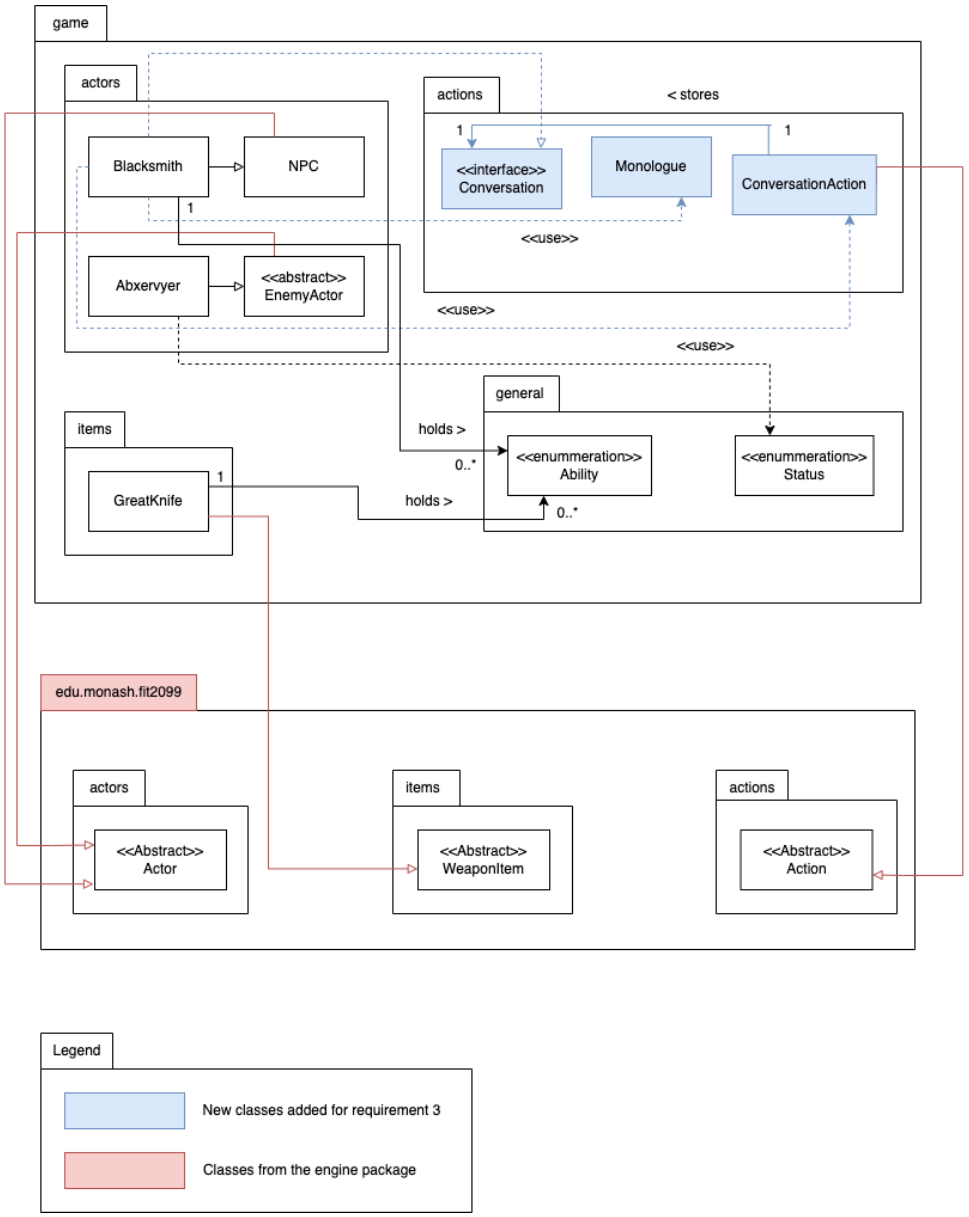
We began this requirement by adding a new 'Blacksmith' class that extends off of the abstract class 'NPC', aligning with the Single Responsibility Principle (SRP). Just like other actors added before it, this new 'Blacksmith' class ensures that all actions and properties of a blacksmith can be handled within its own class. Having different classes that represent different types of actors ensures that we are following the Open/Closed Principle (OCP) as when new actors are added, we do not have to modify the code of any existing actors. This has also allowed us to add the new 'Blacksmith' class very efficiently as we did not have to touch or modify any other class to create this actor. Continuing this technique throughout our design allows our code to be easily extendable in the future.

Additionally, rather than extending from the abstract class 'Actor' we have chosen to create an additional abstract class (that does extend from 'Actor') called 'NPC'. This allows us to separate the types of actors we have within our code whilst ensuring that common attributes between different actor types only have to be implemented once in these abstract classes, further following the DRY principle. The addition of this class also allowed us to delete the 'playTurn' method for both the blacksmith and the traveller. This also aligns with the Liskov Substitution Principle (LSP) as we are able to rely on the 'NPC' class for shared behaviours whilst all subtypes are able to retain all functionality.

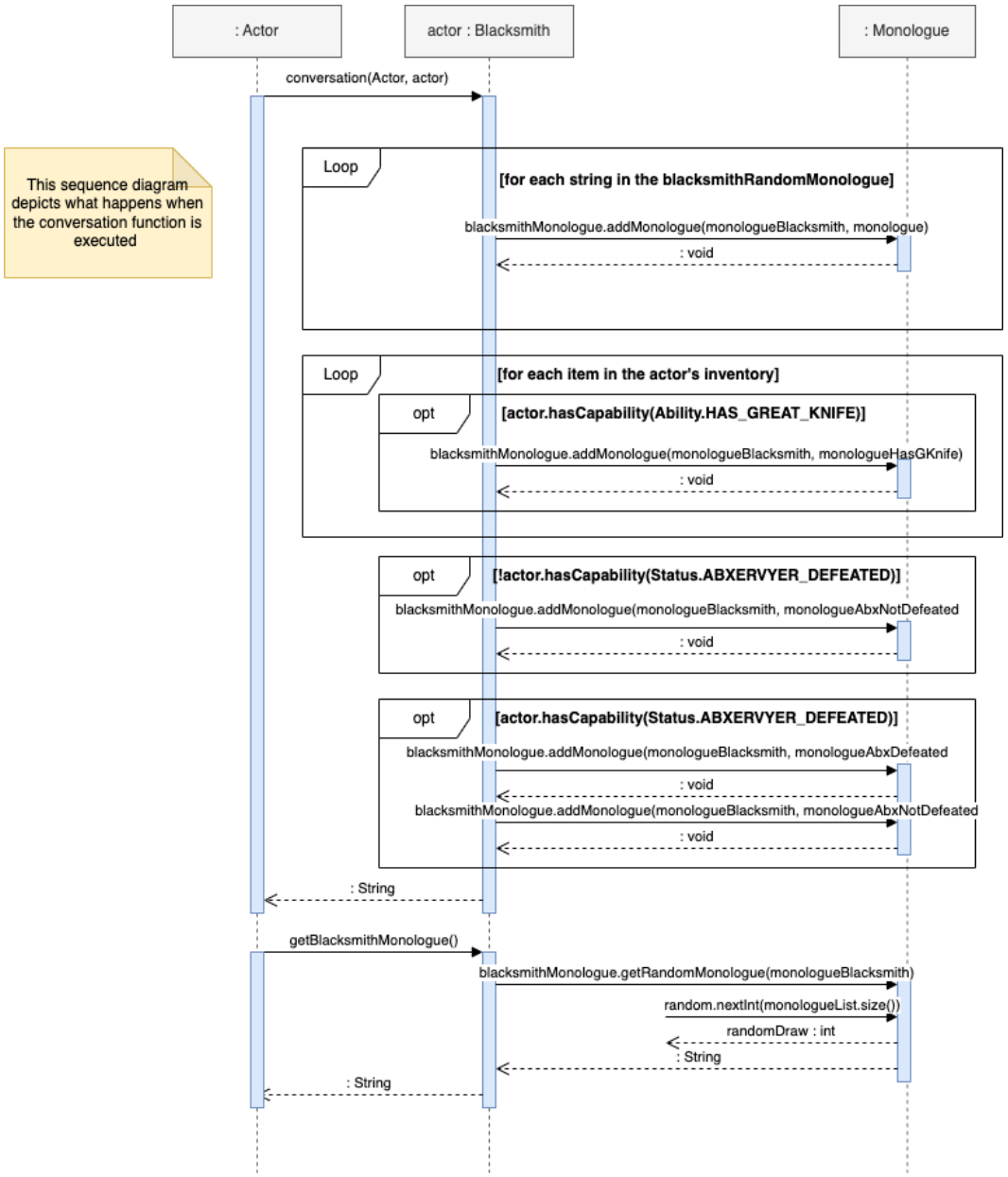
Furthermore, we have created a new 'UpgradeAction' class. This class is solely involved in upgrading different items within our game, also aligning with the Single Responsibility Principle (SRP). The 'Upgradable' interface defining a single method 'upgrade' follows the Interface Segregation Principle (ISP) by ensuring that classes implementing the interface are not forced to utilise methods that they do not use.

Nonetheless, by including so many interfaces and abstract classes, we run the risk of over-engineering our code. Especially with abstract classes such as the 'NPC' class that only includes one common method, though making our code more extensible and maintainable, may also make our code harder to understand and add more complexity. Still, in our opinion, the inclusion of such a class is still the best way to create code that truly aligns with SOLID and DRY principles.

Req 3 UML Diagram



Req 3 Sequence Diagram



Req 3 Design Rationale

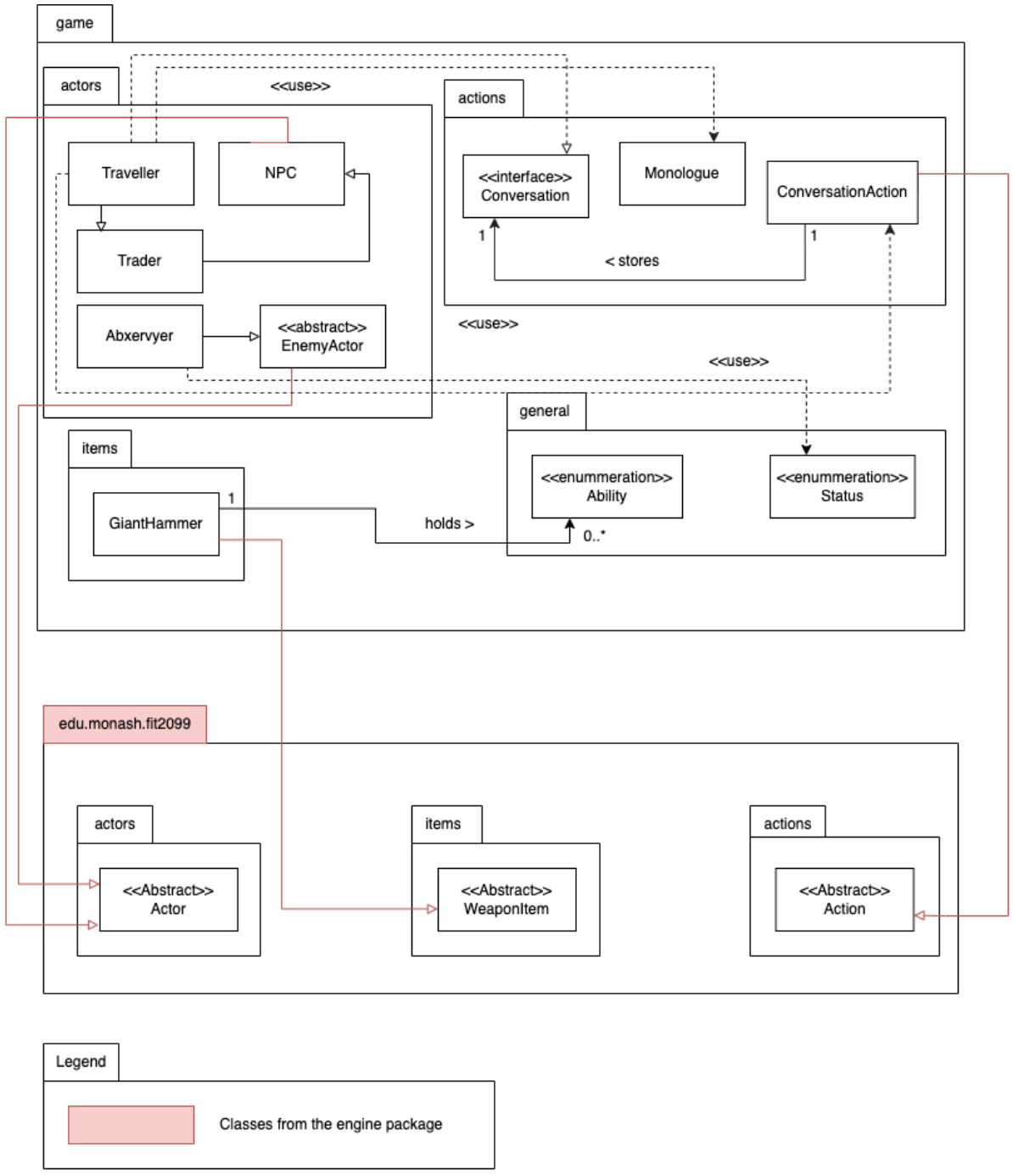
For requirement 3, our design aligns with the Single Responsibility Principle (SRP) as classes are created to have one responsibility. The Monologue class is created to have one responsibility. Its sole purpose is to deal with an array list of conversation options between the Player and another actor such as the Blacksmith. This also means that if another actor wants to converse it can easily create an instance of the Monologue class without needing to create or extend from additional classes.

In addition to the SRP, almost all the classes in the game package extend from an abstract class which aligns with the Liskov Substitution Principle (LSP). The Action abstract engine class for example, is the blueprint for subclasses which extend from it such as ConversationAction. This means the subclass can be used wherever the base class is used without needing to make any changes. This is due to the fact these classes have the same structure and functions which can be overridden if

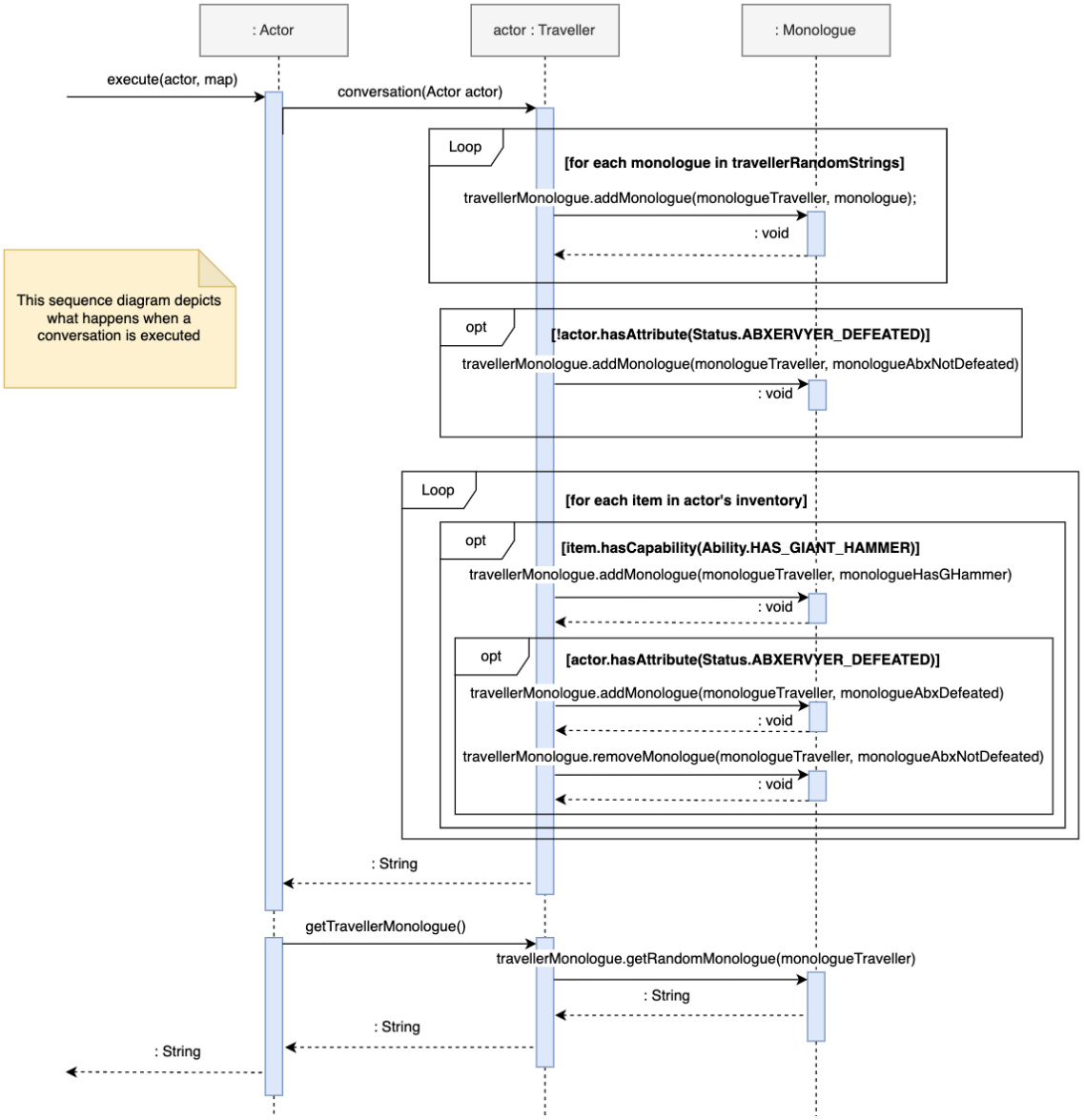
necessary. This ensures the code is easier to extend in the future. The only downside is a class can only extend from one abstract class which can limit the types of functions it can use.

Finally, the Conversation interface follows the Interface Segregation Principle (ISP) as it is a small interface which only contains one function which is to converse between two actors. An interface is useful as a single class can implement as many interfaces as they require and small interfaces can continue following SRP by having a single purpose. However, implementing the interface can be expensive and the contents of the conversation function may become repetitive the more it is used.

Req 4 UML Diagram



Req 4 Sequence Diagram



Req 4 Design Rationale

The design for requirement four uses a lot of the same classes as the design for requirement 3. The traveller extends the Conversation interface which builds the monologue for the traveller and checks the enums of the player to see which lines they can say. This interface as mentioned in the rationale for 3, satisfies the Interface Segregation Principle (ISP) as it contains only one method so other actors that might need to implement it won't be inundated with methods they don't use. However, implementing the interface can be expensive and the contents of the conversation function may become repetitive the more it is used.

Finally, the decision to use an ArrayList to store the Strings, makes the design more extendable as it has an addAll method so it might not necessarily make the developer repeat the addMonologue method when adding a large number of Strings to the possible monologue options which adheres to DRY.

Req 5 UML Diagram

Req 5 Sequence Diagram

Req 5 Design Rationale