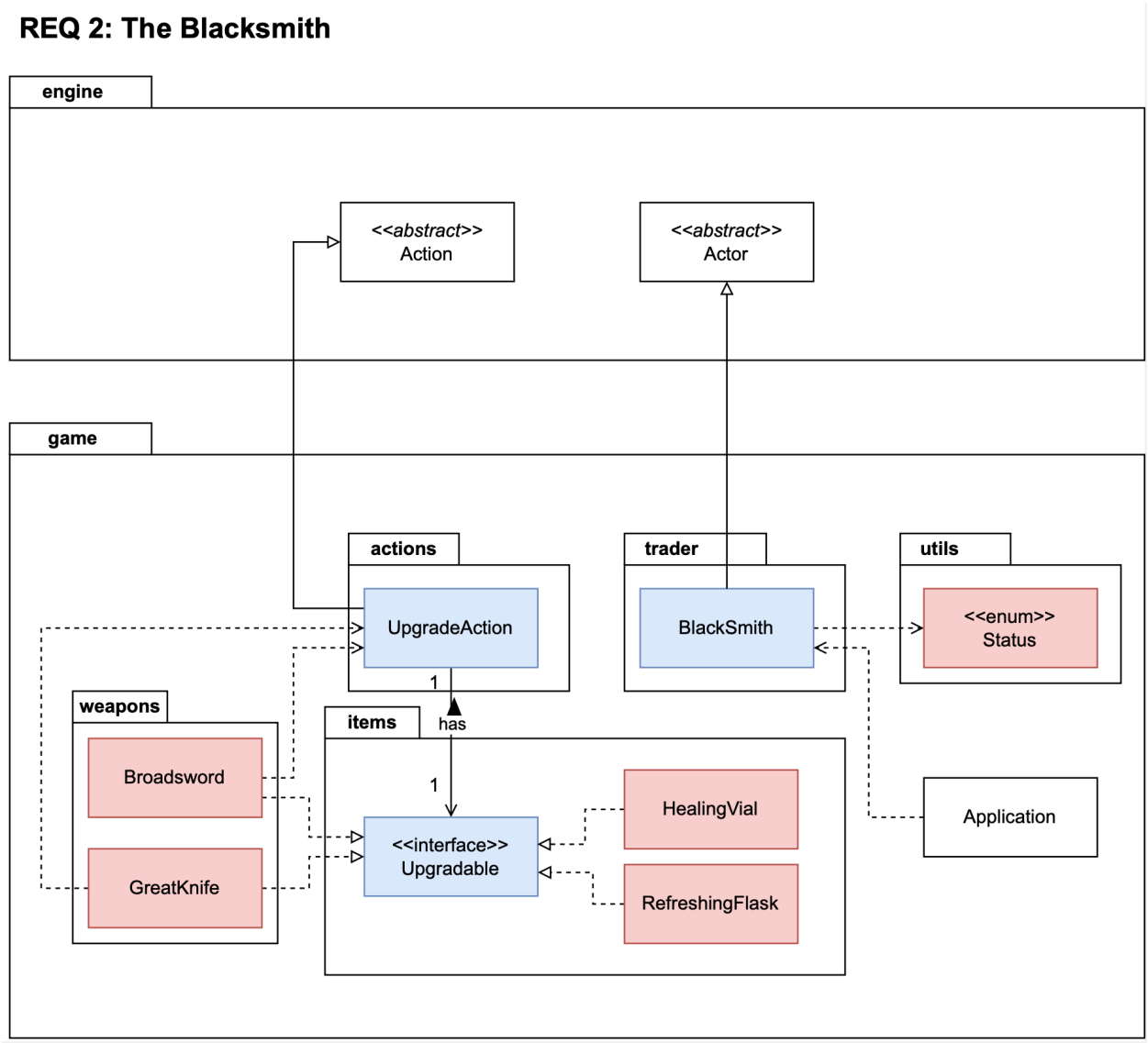
Design Rationale (REQ2)

UML Diagram



Classes Added

- Class Blacksmith
- Interface Upgradable

Class Blacksmith

This class was created to represent the Blacksmith (Actor). In the method **playTurn**(), a DoNothingAction is created to maintain the static position of this actor. In this class's constructor, an attribute Status.IS_MAKER is added to acknowledge Upgradable items that this actor is a Blacksmith (in this particular assignment).

Design Principles Followed

1) Single Responsibility Principle (SOLID)
   - The Blacksmith class primarily represents a specific actor in the game. It encapsulates behavior related to the Blacksmith such as its static position, its attribute (Status.IS_MAKER), and the actions it can perform (to be introduced in REQ3, 4).
2) Open-Closed Principle (SOLID)
   - This class is open for extension (can be subclassed to create variations) but closed for modification. The core behaviors can be extended by overriding methods, but the core structure remains unchanged.
3) High Cohesion
   - High cohesion is maintained in this class. All methods and properties within this class revolve around a single concept: defining the behaviour and attributes of Blacksmith.

Cons of the Design

- This class has the same method implementation of **playTurn**() as the Traveller class (returning a DoNothingAction), creating a duplication. However, we have concluded that creating an abstract class dedicated to solely representing the commonality of staying still does not compensate for the additional system complexity of a new layer of abstraction.

Interface Upgradable

The Upgradable interface represents specific items in the game that are capable of being upgraded. This interface contains two methods: **getUpgradePrice**() and **upgrade**().

The **getUpgradePrice**() method is made to return an integer value indicating the number of Runes required to upgrade the item. The **upgrade**() method is designed to change attributes of the class being called from, according to its upgrade specification.

Design Principles Followed

1) Single Responsibility Principle (SOLID)
   - The Upgradable interface follows the Single Responsibility Principle as it has one responsibility: to define instructions for items that can be upgraded in the game. The interface ensures that each class that implements it only focuses on the functionality of being upgraded.
2) Interface Segregation Principle (SOLID)

- The Upgradable interface ensures that only the classes that implement it provide a specific functionality i.e., getting the price for the upgrade and how an item should be upgraded (changing attributes). Items that are not upgradable are not forced to implement such methods.

3) High Cohesion
   - The Upgradable interface promotes high cohesion by grouping related instructions (returning of price, changing in class attributes), ensuring that all methods defined in the interface are aimed at achieving the functionality.
4) Low Coupling
   - This interface promotes low coupling as it allows flexible implementations. Classes that implement this interface are not tightly bound to specific implementations, enhancing modularity and ease of maintenance.

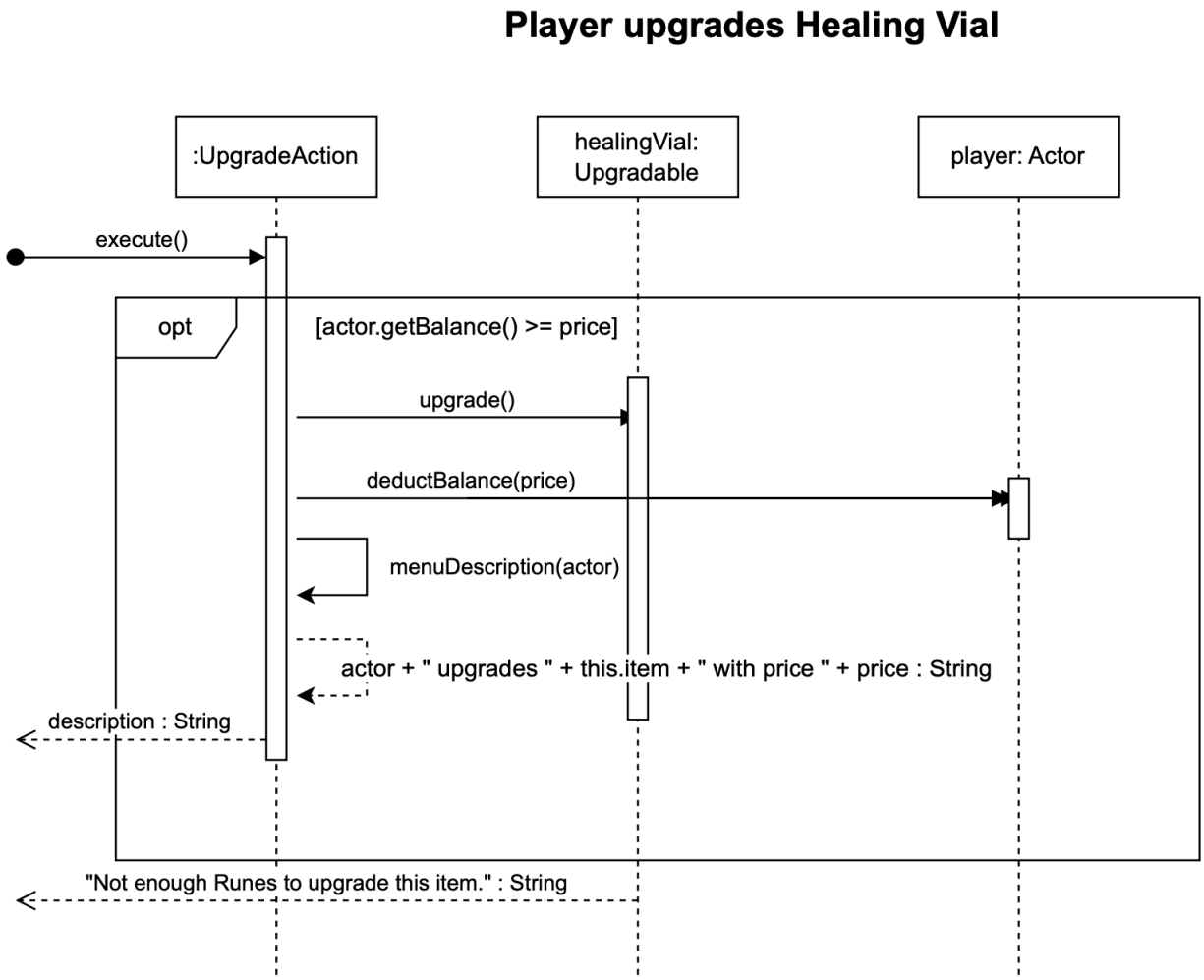
Cons of the Design

1) System Complexity (Violation of SRP)
   - The addition of a new abstraction layer Upgradable adds some complexity to the system as any classes that implement these interfaces are forced to implement its methods, as well as to have additional attributes. This somewhat violates the SRP as each item class has to deal with more than one responsibility: methods related to upgrades on top of returning relevant actions to the player.

Class Removed: Abstract Class Trader

We have decided to remove the abstract class Trader and the attribute Status.IS_TRADER from the previous implementation due to the lack of common features: although both the Traveller and Blacksmith classes perform reducing the player's wallet balance, the difference in the nature of the services they provide (SellAction and UpgradeAction) did not suit the needs of creating an abstract class to group them. Additionally, the abstract class turned out to be empty (with its only functionality being returning DoNothingAction), therefore the abstraction layer has been removed to maintain the system's simplicity as it was thought that the benefits of following the Open-Closed Principle (SOLID) are not strong enough to compensate for the addition of system complexity.

Sequence Diagram for Requirement 2



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 Upgradable interface as both Blacksmith and UpgradeAction extend 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 multiple classes representing the upgradable items implement common methods (getUpgradePrice, upgrade) specified by the Upgradable interface, each method implements instructions that are specific to the item that the class represents.

3. **Can you think of a potential design alternative? What would be the cons of the alternative design?**
   A possible alternative design could be centering the responsibility of checking the type of each item and returning an UpgradeAction accordingly, for example, in the Blacksmith class. This would eliminate the need for each item class to implement the methods in the Upgradable interface. However, this would not only violate the SRP as the Blacksmith class is given more than one responsibility (to store the upgrade specifications and validate instances), but also violates the principle of object-oriented programming.

4. **What design did you end up using?**
   We opted for the design involving an Upgradable interface to be implemented by upgradable items, as demonstrated in the provided code. The design choice promotes code readability, maintainability, and ease of extension, aligning with the principle of object-oriented programming.