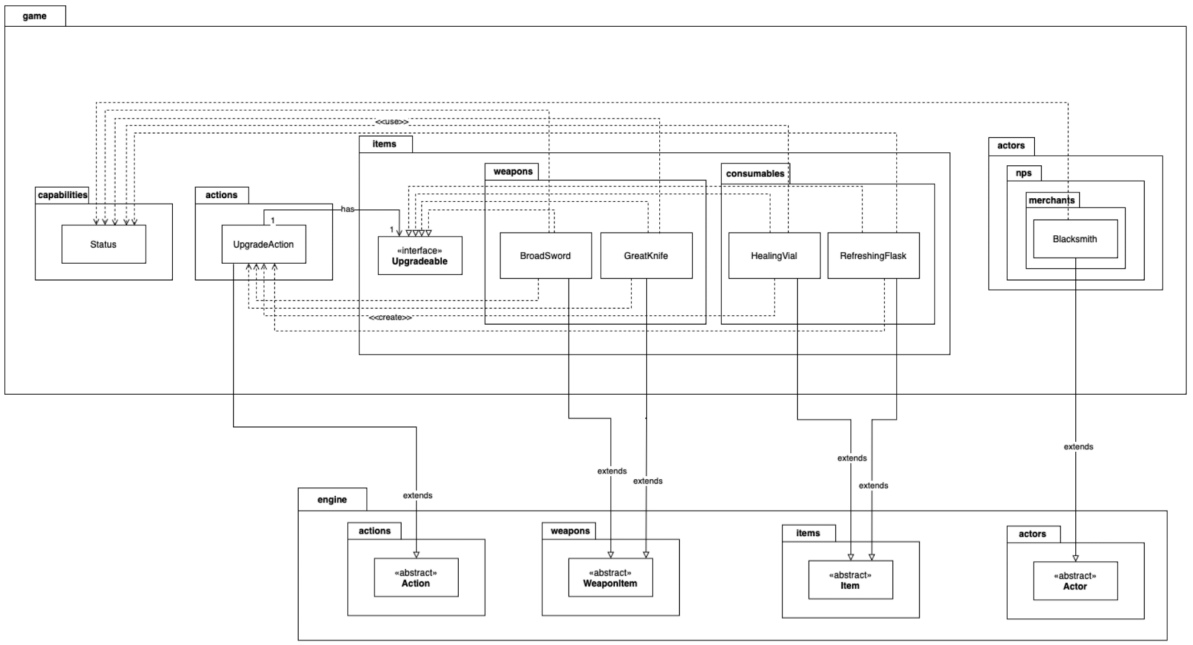

The main objective of this requirement is to add the feature of upgrading items and weapons when the player meets the Blacksmith.

To achieve this, we created an ‘Upgradable’ interface that is implemented by all items and weapons that can be upgraded. This interface includes the methods ‘upgrade()’, ‘getUpgradingPrice()’ and ‘ableToUpgrade()’. The logic of buffing the attributes of the items is done within their respective ‘upgrade()’ methods. This adheres to the Interface Segregation Principle (ISP), as we created a specific ‘Upgradable’ interface that contains only the methods relevant to the implementing classes, and do not burden other classes with superfluous methods that they don’t need.

We also added the ‘UpgradeAction’, which takes in an ‘Upgradable’ as an argument in its constructor, and has an ‘Upgradable’ as its attribute. We call the Upgradable’s upgrade method inside the ‘execute’ of ‘UpgradeAction’. This adheres to the Dependency Inversion Principle (DIP), as we rely on the abstraction of the ‘Upgradable’ rather than its specific implementation. This also adheres to the Single Responsibility Principle (SRP), as ‘UpgradeAction’ has a clear and focused responsibility of upgrading items.

HealingVial, RefreshingFlask, Broadsword and GreatKnife, can be used interchangeably in contexts where an ‘Consumable’ is expected, like when passing the argument through the ‘UpgradeAction’ constructor, this demonstrates adherence to the Liskov Substitution Principle (LSP).

This implementation also adheres to the Open/Closed Principle (OCP), as we can easily add more entities that can be upgraded by the player without modification to the existing code. We just have to make these classes implement the ‘Upgradable’ interface.

Advantages
Extensibility

Our implementation is easily extensible for future upgradables that might be added. Future upgradable items can just implement the ‘Upgradable’ interface without requiring modifications to existing code, this adheres to the Open/Closed Principle (OCP).

Modularity
The use of the 'Upgradable' interface and the 'UpgradeAction' class promotes modular design. Each upgradable item encapsulates its own upgrade logic, making it easier to understand and maintain.

Reusability
With the 'Upgradable' interface, the upgrade logic is reusable across different upgradable items and weapons. This promotes code reuse and reduces redundancy and avoids repetitions (DRY).

Disadvantages
Maintenance Overhead
Ensuring that all classes adhere to the 'Upgradable' interface and making updates as the codebase evolves can require additional maintenance effort.

Performance Overhead
Depending on the language and runtime, the use of interfaces and abstractions can introduce a slight performance overhead due to dynamic method dispatch. In performance-critical applications, this might be a concern.