Design Rationale for Requirement 3

Traveller
This requirement adds a new type of NPC who is a trader. It is assumed that there can be different types of traders so an abstract Trader class is created. It is also assumed that all traders cannot move and cannot be attacked by any other actor. Therefore, the abstract Trader class returns a doNothingAction for every turn.

The purpose of traders in this game is to allow players to buy new items and sell items from their inventory. There is an assumption that instances of the same item have a fixed selling price to all types of traders but a different price at which they would buy the item. Also all the items have the same pricing strategy and type of scam for both buying and selling across all types of traders.

The trader type implemented in this requirement is a Traveller. It extends from the abstract Trader class and implements the allowableActions() method. The Traveller's implementation will be discussed in the Trading section below.

Since it is assumed that the Traveller has an unlimited supply of items that they can sell, a list of items that can be bought by the player is updated at every turn. At this point, the Traveller could sell three types of items: the Healing Vial, the Refreshing Flask and a Broadsword.

This implementation completely aligns with the Open-Closed Principle, since new types of traders can be easily implemented in the future through the abstract class. The only disadvantage of this implementation is that the pricing strategy and scam type will be the same for an item across all traders.

Trading
BuyAction and SellAction
There are two new actions implemented in this requirement in order to support traders: BuyAction and SellAction. Both of these extend from the abstract Action class. There are two possible implementation methods for these classes which will be discussed in this rationale.

First implementation
The first implementation of this method had both BuyAction and SellAction defined inside the Traveller's allowableActions() method. A new BuyAction method was added to the allowable actions list of the trader, creating a new instance of the item to be sold. To allow the Player to sell the item, the Traveller's allowable actions looped through the Player's inventory to create a new SellAction with every inventory item that had a capability SELLABLE. This approach did not align with the Single Responsibility Principle, as it loops through the Player's inventory and creates an action for every item (the better approach to this is that each item creates its own list of actions).

Also new interfaces were created: Buyable and Sellable, that are passed down as a parameter from a BuyAction and a SellAction. These interfaces had methods getPrice() and getScamChance(), and are implemented by items that can be bought or sold. This perfectly aligns with the Interface Segregation Principle.

Buy and sell prices were implemented as attributes inside each of the buyable or sellable items. In this implementation method an abstract ConsumableItem class implemented the Buyable and Sellable interfaces, assuming each Consumable item could also be bought and sold. This does not satisfy the Single Responsibility Principle (ConsumableItem class is only supposed to be responsible for ConsumeAction) as well as the Open/Closed principle (for example, if the item can be consumed but cannot be bought). This also leads to a Code Smell, since it ends up with too many arguments in the ConsumableItem constructor (name, display character as well prices and scam values for both buy and sell). This was another reason to remove the abstract ConsumeAction class in Requirement 2 and add a Consumable interface to each of the items instead. Furthermore, this implementation breaches the assumption that different types of traders would sell items with different prices.

The main disadvantage of this implementation is that BuyAction and SellAction have too many responsibilities: they decide if the scam was successful or unsuccessful as well as type check these scams using if-else statements. Moreover, these actions use castings to add or remove items from the actor’s inventory.

Second implementation
The second implementation method had BuyAction implemented inside the Traveller class and SellAction implemented inside each of the items’ classes. The Traveller had the items’ selling prices as static attributes, adding new BuyActions for each of the items the Player could buy from the Traveller. SellAction was added to the allowable action list of the Items, with each item having their sell price as a static attribute (since it is assumed that the sell price is the same across all the traders as mentioned before).

Each item would now directly implement the Buyable and/or Sellable interfaces, instead of implementing them inside the abstract class. Buyable and Sellable interfaces’ methods were changed to be bought() and sold() respectively. The main difference from the previous implementation is that these methods do all the work inside each of the items. For example, the bought() method for the Healing Vial adds an instance of this item to the Player’s inventory, deducts balance from the Player and adds balance to the Traveller. BuyAction only calls this method on the item.

This approach follows the Single Responsibility Principle since each item is responsible for their own implementation of buying and selling. Furthermore, this allows the implementation of different types of scam which also leads to the Open-Closed principle by allowing different scam types. So this method was used in the final implementation.

Both BuyAction and SellAction for the Healing Vial were represented in two separate Interaction Diagrams, as these are examples of the major interaction implemented in Requirement 3.

Change of price
As traders may increase or decrease the buy or sell prices of the items randomly at each turn, methods were created to implement this feature, as discussed in the following section.

First implementation
The first implementation used pricing strategy classes. A new Pricing interface was created and implemented by three different pricing strategies: IncreasedPrice, ReducedPrice and RegularPrice. These pricing classes had a method to change the price if the probability of a scam was successful, or else would return the original price.

This implementation had issues following the Dependency Inversion Principle, as in some cases, concrete item classes were creating and executing concrete pricing strategies ,instead of depending on the Pricing interface. In order to follow this principle, the pricing strategies should be passed as parameters to the BuyAction and SellAction and executed there. But this method increases the number of arguments leading to the Code Smell. Moreover, this approach creates unnecessary dependencies and associations between items and pricings.

Second implementation

The second attempt at implementing this feature created static methods inside the Utility class. These methods achieved what the pricing strategies in the first implementation did: calculating new prices if the probability is successful or just returning the original price. These methods are called inside the bought() and sold() and overall make the code easier to understand and more readable. This method was used in the final implementation.