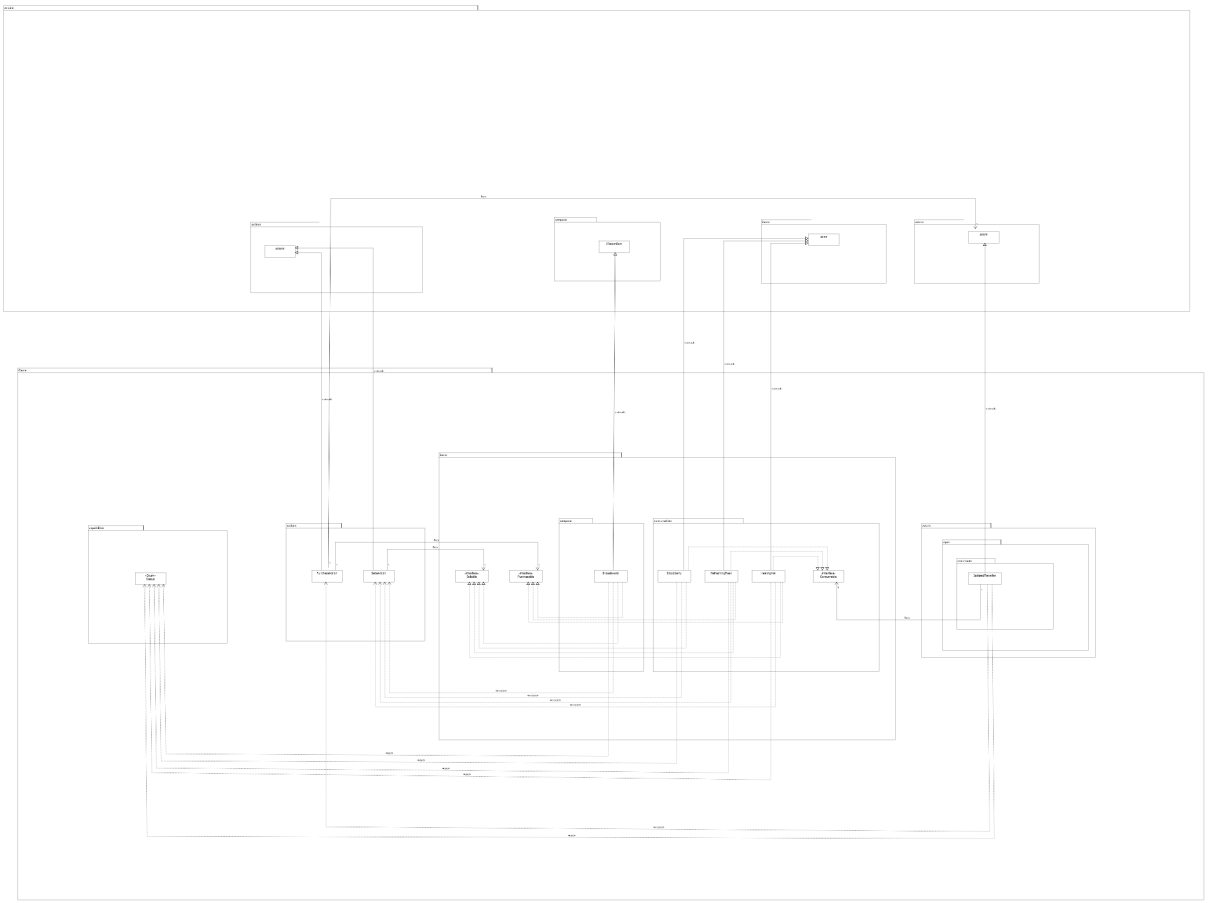

The main objective for REQ3 is to implement the NPC ‘IsolatedTraveller’ that can sell and purchase stuff from the Player.

To do this, we decided to make a ‘Purchasable’ and ‘Sellable’ interface, which will be implemented by items that have these capabilities. This implementation adheres to the Interface Segregation Principle (ISP) as items will only have to implement the specific interface that they need and not be burdened with superfluous methods. For example, items like ‘Bloodberry’ can only be sold to the IsolatedTraveller but not be bought from him, this means that Bloodberry only needs to extend the ‘Sellable’ interface.

‘Purchasable’ interface has a ‘purchase’ method that accepts 2 actors as its arguments, the player and the seller, and a ‘getPurchasePrice’ method that accepts the seller Actor. With these methods, we are able to have different purchase prices for different sellers. ‘Sellable’ interface has a ‘sell’ method. Both ‘purchase’ and ‘sell’ methods modify the Player’s inventory and balance when purchasing or selling something, they are called in the execute method of ‘PurchaseAction’ and ‘SellAction’ respectively.

‘PurchaseAction’ and ‘SellAction’ constructors take in instances of ‘Purchasable’ and ‘Sellable’ respectively. This implementation is similar to REQ2’s implementation, in which the ‘ConsumeAction’ constructor takes in an instance of ‘Consumable’ as an argument. This implementation adheres to the Dependency Inversion Principle (DIP) as these actions depend on the abstraction of ‘Purchasable’ and ‘Sellable’ items rather than their detailed implementations.

Furthermore, items that can be bought from the IsolatedTraveller (HealingVial, RefreshingFlask and BroadSword) can be used interchangeably in contexts where an ‘Purchasable’ is expected, for example: when passing the argument through the ‘PurchaseAction’ constructor, this demonstrates adherence to the Liskov Substitution Principle (LSP).

Alternative implementation:
Originally, the idea was to have everything be done in the IsolatedTraveller’s PlayTurn, in which we would loop through both the IsolatedTraveller’s inventory’s items as well as the Player’s. However, we quickly realised that this would involve downcasting the Player’s inventory items from ‘Item’ to ‘Sellable’ in order to pass them as arguments into ‘SellAction’s’ constructor. This means we would be downcasting to a more detailed layer (a lower level abstraction), which is not ideal as we reduce the extensibility and reusability of the system. As such, we decided to shift the responsibility of returning ‘SellAction’ to the items’ respective AllowableActions instead.

Advantages
Future extensibility
This implementation allows for easy extensibility as more items are added in the future. New items can implement either ‘Purchasable’ or ‘Sellable’, or both. This adheres to the Open-Closed Principle (OCP), as we can easily add more entities that can be bought and sold by the player without modification to the existing code.

Avoidance of Downcasting
Our implementation avoids the use of downcasting, which is beneficial for maintainability, reduced risk of runtime errors, and increased flexibility.

Disadvantages
Complexity and Development Time
In order to avoid the use of downcasting, we had to find a workaround that was quite a bit more complex and less straightforward. Which took more development time as well.

Testing Complexity
We took much more time testing and debugging our code compared to if we had just used downcasting instead.