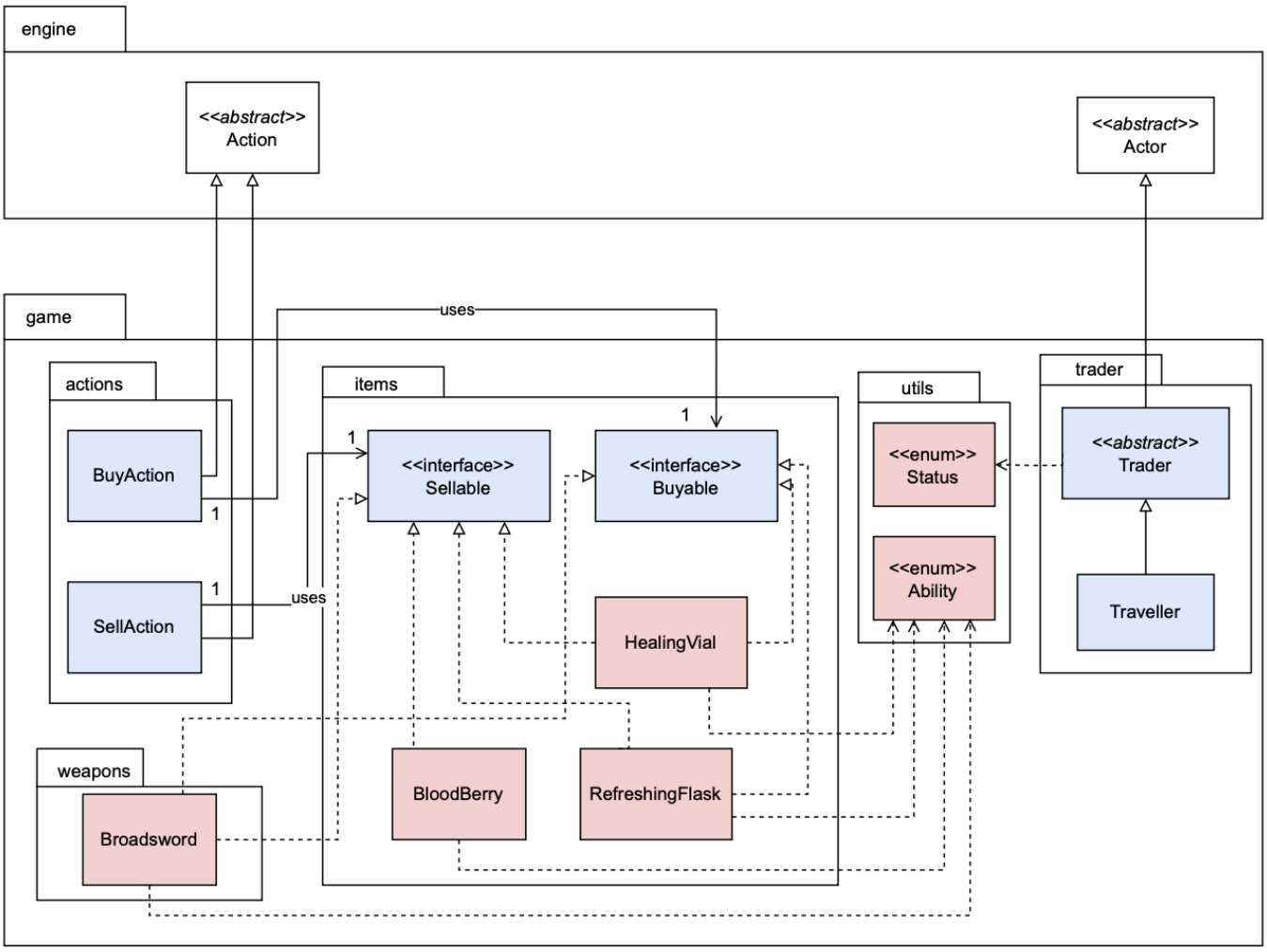
Design Rationale (REQ 3)



Changes Made from Assignment 1

No major changes were made from Assignment 1. All the code used follows the design principles, and expandable for the requirements in Assignment 2.

Implementation

Classes Added

1. Abstract Class Trader
2. Class Traveller extends Trader
3. Interface Buyable
4. Interface Sellable
5. SellAction extends Action
6. BuyAction extends Action

Codes Modified

1. For each Buyable item (HealingVial, BroadSwrod, GreatKnife) added:

    - Interface Buyable's method overrides: addToInventory, transactionSuccess, getBuyPrice.
2. For each Sellable item (BloodBerry, HealingVial, RefreshingFlask, BroadSwrod, GreatKnife, GiantHammer) added:
    - Interface Sellable's method overrides: removeFromInventory, getSellPrice.

Design Principles

Interface Buyable, Sellable

The Buyable interface was created to represent specific items that are purchasable from traders, and the Sellable interface represents specific items that can be sold to the traders.

Design Principles Followed

1. **Interface Segregation Principle (SOLID)**
    - The Buyable interface ensures that the classes that implement it provide a specific functionality, i.e, adding the purchased item to the player's inventory. The Sellable interface ensures that the classes that implement it provide a specific functionality, i.e, getting the item price, removing the item from the player's inventory, and getting the original price. This allows flexible extension of new buyable items.

Abstract Class Trader

This abstract class was created to represent any type of traders.

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - The Trader abstract class is responsible for managing common functionalities shared by any trader objects. This includes addition of capability IS_TRADER and maintaining the action of not moving.
2. **Open-Closed Principle (SOLID)**
    - The Trader class adds a new layer of abstraction to represent all traders. This class contains all required attributes that non-player characters have (IS_TRADER Status, DoNothingAction instance). This means any extensions of the system involving the addition of new types of traders can be achieved by the addition of new classes that extend the Trader class without a need to modify the Trader class.
3. **Liskov Substitution Principle (SOLID)**
    - The subclass (Traveller) is substitutable for its super class (Trader). This is evident in methods like allowableActions where the subclass calls the superclass's method via super.allowableActions() and then add their unique behaviours (DoNothingAction).
4. **DRY (Don't Repeat Yourself)**
    - The common functionalities and properties among traders are kept in the Trader class, preventing repetitions within its child classes.
5. **High Cohesion**
    - The Trader class focuses solely on functionalities related to traders. This design ensures that related functionalities are bundled together in a logical manner.

Class Traveller

This class was created to represent Travellers.

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - The Traveller class has a single responsibility: creating an instance of a Traveller and handling its action.
2. **Open-Closed Principle (SOLID)**
    - Both classes are open for extension (we would extend them to add more initialization parameters for the enemies in the future), but closed for modification (we won't have to change the classes when adding new types of enemies)
3. **Don't Repeat Yourself (DRY)**
    - BuyAction and SellActions are created according to the items that the Traveller provides. Any changes in such list of items will not necessitate the changes in other classes, preventing code repetition.

Pros of the Design

1. **Abstraction**
    - The introduction of the abstract class Trader not only makes the system easy to comprehend, but also to be extended and managed. Future additions of new trader types can be done by extending this class and any updates in common funcitonalities in traders can easily be implemented by modifying this class.
    - Addition of the interfaces: Buyable and Sellable introduces a new layer on items on top of their parent class. This improves intuitiveness and extensibility of the system as new items that are either purchasable or sellable by the player can implement these interfaces to represent such capabilties.

Cons of the Design

1. **Complexity**
    - The addition of a new abstraction layer (Buyable, Sellable) 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 a class has to deal with more than one responsibility: methods related to transactions on top of returning relevant actions for the player.