Requirement 3 Design Rationale

Implementation

JarOfPickles class: An consumable item which extends Item abstract class and implements Consumable interface. It is an item which can be picked up by a player and also can be consumed. It implements the consume logic and execution in the consume method from Consumables interface then a ConsumeAction is added in allowableActions. Players who consume this item have 50% of healing / hurting by 1 point. After consuming, this item will be removed from the player's inventory.

PotOfGold class: Also a consumable item which extends Item abstract class and implements Consumable interface. It can be picked up and consumed by the player. The consume action is taking out the gold bar from the pot and adding $10 to the player's wallet. It is just done by adding a $10 amount in the consume() method and the Gold Pot being removed. ConsumeAction handles the execution of the logic and is added to the ActionList in allowableActions.

Puddle class: Puddle ground class is modified by adding an inheritance of Consumables interface. Drinking water action execution and logic to increase maximum health point is achieved by implementing an abstract consume method and adding a ConsumeAction into the allowableActions return list. Overriding allowableActions method has an if statement to check if the player is on a puddle ground, only it can consume water on the exact location it is standing on. In the consume method, the maximum health point is increased by 1 by calling the actor.modifyAttributeMaximum method and returning a string msg which indicates the result of consuming water.

Modification

ConsumeAction class: Previously was a HealingAction in Assignment 1 which includes the actor.heal() action in execute. This is a bad approach for being too specific and not universal. It was then modified to a ConsumeAction to better encapsulate general consume action logic by removing the healing execution. It is also modified by removing the actor.removeItemFromInventory line in the execute method. Which makes ConsumeAction able to handle more situations and be more flexible. Lastly, ConsumeAction's execute method is only calling the consumable consume method which is where the action will be executed.

LargeFruit and SmallFruit class: Just HealingAction changed to ConsumeAction and added the actor.heal() healing execution in the consume method for both classes.

Player class: added a display.println("wallet balance: $" + this.getBalance()) line to display the wallet balance of a player which will be used in requirement 4 later.

Implementation approach

First of all, the fruit healing action was executed by a HealingAction in our base code in assignment1. It wasn't a good approach so we modified this action to a general ConsumeAction which we will be using to be able to handle all general consume actions. The exact healing execution in this action was removed, making it a consume action which all classes that implement the Consumable interface can be executed in the execute method which will call the consume method, and we continued our work with ConsumeAction.

There are two new items to be created which have the same logic but slightly different execution. So for the first item, a Pot of Gold, the requirement states that the pot of gold can be picked up by the player then choose to take the gold out and add balance to the wallet. Which also, we can see this action as consuming the gold pot. Once the player consumes the gold pot (takes out the gold from the pot), the player will have an amount of money being added to the player's wallet (consume execution), then this pot of gold will be stated as consumed by the player (remove from the player's inventory). Even though it didn't explicitly mention in the requirement that the player can "consume" the pot of gold, from the logic and execution, we know that this is actually just a consume action. So, this item implements the Consumables interface to implement the consume method and a ConsumeAction is added into the ActionList in allowableActions method. So, this entire consume action flow and logic was easily done by implementing the previously created interface (Consumables) and action (ConsumeAction) which made the code extension implementation simple and efficient! The second item to create is a jar of pickles. This item was clearly stated to be able to be consumed by the player. For the base code we had implemented previously, Item abstract class (code given in engine) and Consumables interface (from assignment 1), implementing a consumable item is pretty easy. Same as the just created item PotOfGold , also LargeFruit and SmallFruit we have done in assignment 1, JarOfPickles needs to only extend Item abstract class, implements Consumables interface, implement abstract method then add a ConsumeAction. Two consumable items which we have to newly create are done! For further information, initially I had implemented an abstract class ConsumableItem for all consumable items to reduce the code redundancy and centralise the common logic for consumable items by implementing the allowableAction method with adding a ConsumeAction. However, this would lead to an issue of how if these items want to implement more interfaces? Maybe purchasable or tradeable? The items might still need to implement interfaces and interfaces allows multiple inheritance which encapsulating one interface into an abstract class wasn't needed and it would make the design more complex like allowableActions would have to be overridden very often which violates the SOLID principles. Having an abstract class for consumable items might be too complex and not efficient. So, for all items, extending Item abstract class and implementing interfaces which it can function eg. Consumables / Purchasable / Tradeable by their own logic will be the best approach.

The other feature for requirement 3 is players will be able to drink water from the Puddle ground and increase their maximum health point. Previously, we were only working on consumable items. However, ground can also be consumed! Like this requirement, drinking water is also a consume action! This can easily be implemented with Puddle implementing the Consumables interface and abstract method consume(). For grounds, there is a little bit more to do which is to only let players drink the water from the exact location the player is on. So, checking if the ground location contains that player is needed in the allowaleAction method before adding the ConsumeAction. We do not have to create a new action for drinking water where all consume actions can be handled just with the ConsumeAction which is very efficient and at the moment, all requirements are achieved! However, I did a minor change in the ConsumeAction by removing the line actor.removeItemFromInventory(Consumable item) in the execute method. With this code in the ConsumeAction, we will face an error which grounds cannot be removed. After deleting this line(which will be added in all consumable items consume method itself), the ConsumeAction can handle both consumable ground and item cases! Additionally, removing this line of code is not only to make it able to handle consumable grounds, but also, we can think of how if there is an item which player can always have it in their inventory permanently and can consume it whenever they want? Having the remove item executed in ConsumeAction wouldn’t allow this to happen as it will be immediately removed after being consumed which reduces the flexibility.

Concepts and principles applied

1. Single Responsibility Principle (SRP) : By having the Consumables interface define a minimal contract (consume method), it ensures flexibility and consistency across various consumable entities. JarOfPickles, PotOfGold and Puddle each implement the Consumables interface directly, offering unique consumption behaviours. ConsumeAction relies on the Consumable interface, encapsulating the consumption process and allowing seamless interaction with any consumable entity. This approach adheres to the Single Responsibility Principle (SRP) by separating each entity’s consumption behaviour and action logic.
2. Open / Closed Principle (OCP) : The use of inheritance in JarOfPickles and PotOfGold simplifies common functionality, while Open / Closed Principle (OCP) is achieved since new consumables can be easily added by implementing Consumables or extending Item.
3. Liskov Substitution Principle (LSP) : The Consumables interface is implemented by multiple classes (JarOfPickles, PotOfGold, Puddle) which represent different types of consumable items.Each of these classes can be substituted for another without affecting the system's behaviour. For example, ConsumeAction can consume any object that implements Consumables, regardless of whether it's a jar of pickles, a pot of gold, or a puddle.
4. Interface Segregation Principle (ISP) : ensures that all consumable entities implement a lightweight interface, focusing solely on consumption. The interface declares a single method consume(Actor actor) for handling consumption actions. This ensures that all consumable entities implement this interface without having to implement unnecessary methods not related to consumption.

5. Dependency Inversion Principle (DIP) : The ConsumeAction class depends on abstractions (Consumables) rather than concrete implementations. This is evident in the constructor where it takes a Consumables parameter, which allows for flexibility in the type of consumable being consumed.

Advantages include reduced redundancy by encapsulating common logic, improved maintainability by separating concerns, and enhanced flexibility through abstraction. One minor disadvantage is that items implementing Consumables directly might need to duplicate some logic, like allowableActions. Nevertheless, this design can be easily extended by introducing new consumable items or ground entities, as long as they implement the Consumables interface. For instance, adding a new consumable class like EnergyDrink(In req4) would simply require implementing Consumables and providing specific behaviour in the consume method. This ensures adherence to OCP and allows the system to grow without modifying existing classes. By following these principles, the design achieves modularity, maintainability, and extensibility, aligning well with the OOP principles.