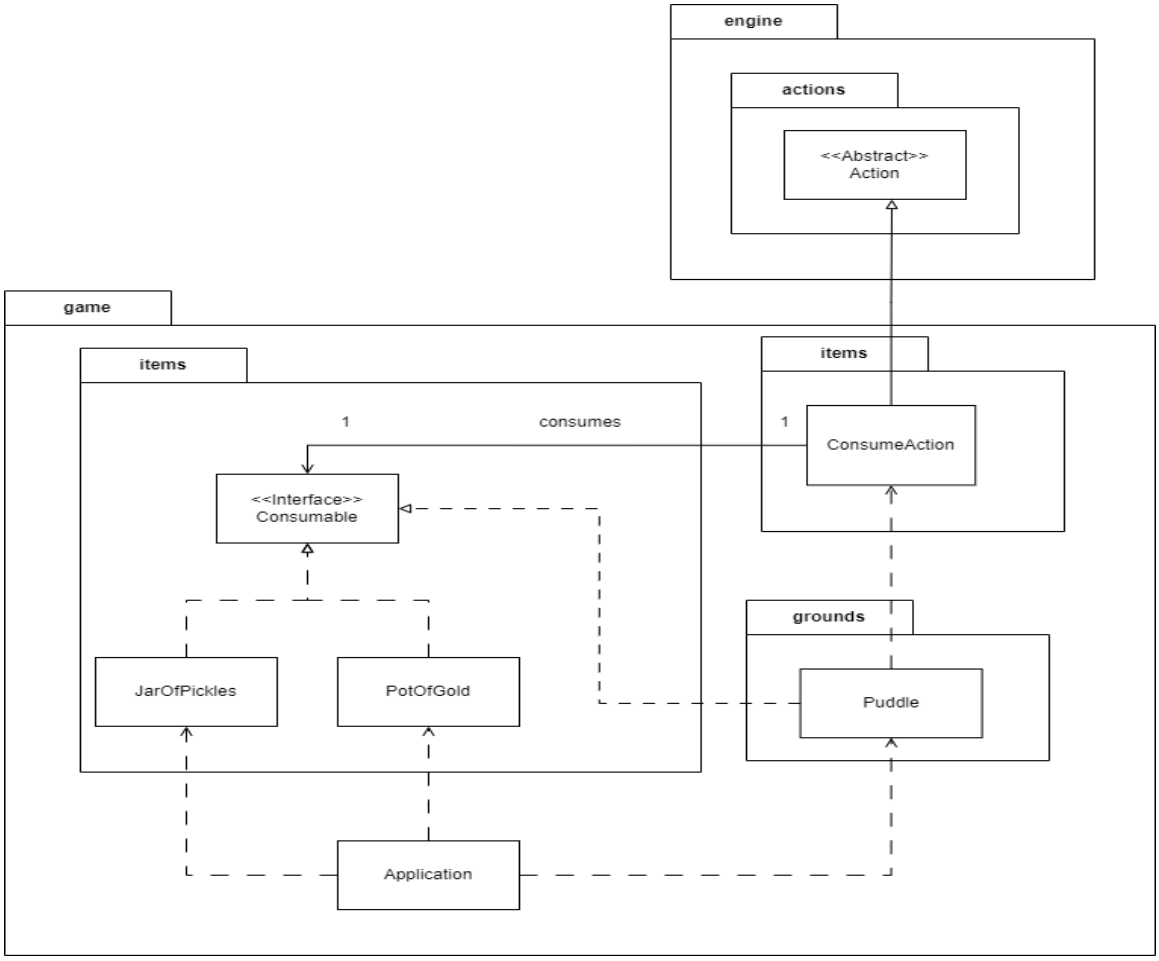
Assignment2 Design Rationale

Requirement 3: More scraps

UML Diagram:



Design Goal:

The design goal for this requirement is to incorporate two new features: adding two new consumable item classes, which are PotOfGold and JarOfPickles, and adding new feature to the puddle while adhering closely to relevant design principles and best practices.

Design Decision:

Before implementing the two new classes, I decide to change the Consumable abstract class to interface. It is possible that there is a subclass of other unconsumable classes but the subclass is consumable. In that scenario, the

subclasses cannot extend the Consumable abstract class so we could only write the same code in the subclass and it violates the DRY principle. Besides that, it affects the ConsumeAction too because ConsumeAction has an association relationship with Consumable abstract class. If we want to consume the object of the subclass, we need to link another relationship between the subclass and ConsumeAction and it violates the Dependency Inversion Principle. Therefore, we decide to change it to interface so the other subclass can implement. Besides that, the extendibility is enhanced since we can let the all types of classes implement the Consumable. However, it might potentially violate the DRY principle because we need to write same code in the SmallFruit class and BigFruit.

Besides that, we decide to modify the implementation of the consumeConsumable method of the consumable item classes and the execute method of ConsumeAction so it can return string in different formats. Now, the description of the effect after consuming the item is done by the consumable item classes. It is logical because only consumable item class “knows” the effect after consuming the item.

In implementing the two new consumable item classes, which are JarOfPickles and PotOfGold. We decide to let these two classes extends the Item class and implement the Consumable interface so we do not need to write the code repeatly(DRY principle). Besides that, it follows the Liskov Substitution Principle. Since JarOfPickles and PotOfGold are the subclasses of Item class, they can be upcasted and replace the object of Item class. Besides that, they implement the Consumable interface so they do not need another relationship with the ConsumeAction because Consumable interface already has a relationship with the ConsumeAction(Dependecy Inversion Principle).

In implementing the new feature of puddle, we choose to let the Puddle class implements the Consumable interface since Puddle is a subclass of Ground class. Therefore, Puddle class can return a list of actions that contains object of ConsumeAction so player can consume the puddle when player is standing on the puddle. However, due to the implementation of engine code, player is able to consume other puddle’s water although they are not standing on the puddle. Therefore, we need to check the location of player is same as the location of the puddle or not. If yes, then add the ConsumeAction to the actionlist, else return an empty actionlist.

Alternative Design:

One alternative approach could involve [describe an alternative solution or design approach]. However, [briefly explain why the chosen design is preferred over the alternative(s), highlighting any advantages or disadvantages]. You can include a snippet on code of the alternate implementation to assist with your explanations within this section too.

We have two alternative designs. First, we do not change the Consumable to interface. We create two classes, which are JarOfPickles and PotOfGold and let these two classes extend the Consumable abstract class. It follows the DRY principle since we do not write the same code inside the JarOfPickles and PotOfGold. Besides that, we do not violate the Dependency Inversion Principle. Since JarOfPickles and PotOfGold extends the Consumable abstract class, we do not need to link these two classes to the ConsumeAction class. For adding new feature to puddle, we create a class called DrinkWaterAction and let it extends the Action abstract class. Same goes here, we do not violate the Dependency Inversion Principle because we do not need to link the DrinkWaterAction to the Actor class. Inside the execute method of DrinkWaterAction, we directly increase the maximum health of player.

Analysis of Alternative Design:

The alternative design is not ideal because it one SOLID principle:

1. **[SRP]:**

    - The DrinkWaterAction is doing the samething as the ConsumeAction. It means that their implementation is similar. Let say if we need to make some changes to DrinkWaterAction or ConsumeAction, we need to make the same changes to other class. It violates the Single Responsibility Principle.

Secondly, the difference between the alternative design 1 and alternative design 2 is the adding the new feature to the puddle. Instead of creating a DrinkWaterAction, we can reuse the ConsumeAction. Besides that, the reason of player can directly drink the water from puddle is he forgets to bring water bottle. In other word, there is a possibility that there is a new item called water bottle and its functionality is collect the water. Therefore, we need a class to represent the water so we create a class calls Water and let it extends the Consumable abstract class.

Analysis of Alternative Design:

The alternative design is not ideal because it violates one principle

1.[Law of demeter]:

- Instead of creating a class to represent the water, we can change the Consumable abstract class to interface so puddle can implement it. Then we do not need to create a class to represent the water. It violates the law of demeter because we have created another class and increase the number of relationships between classes.