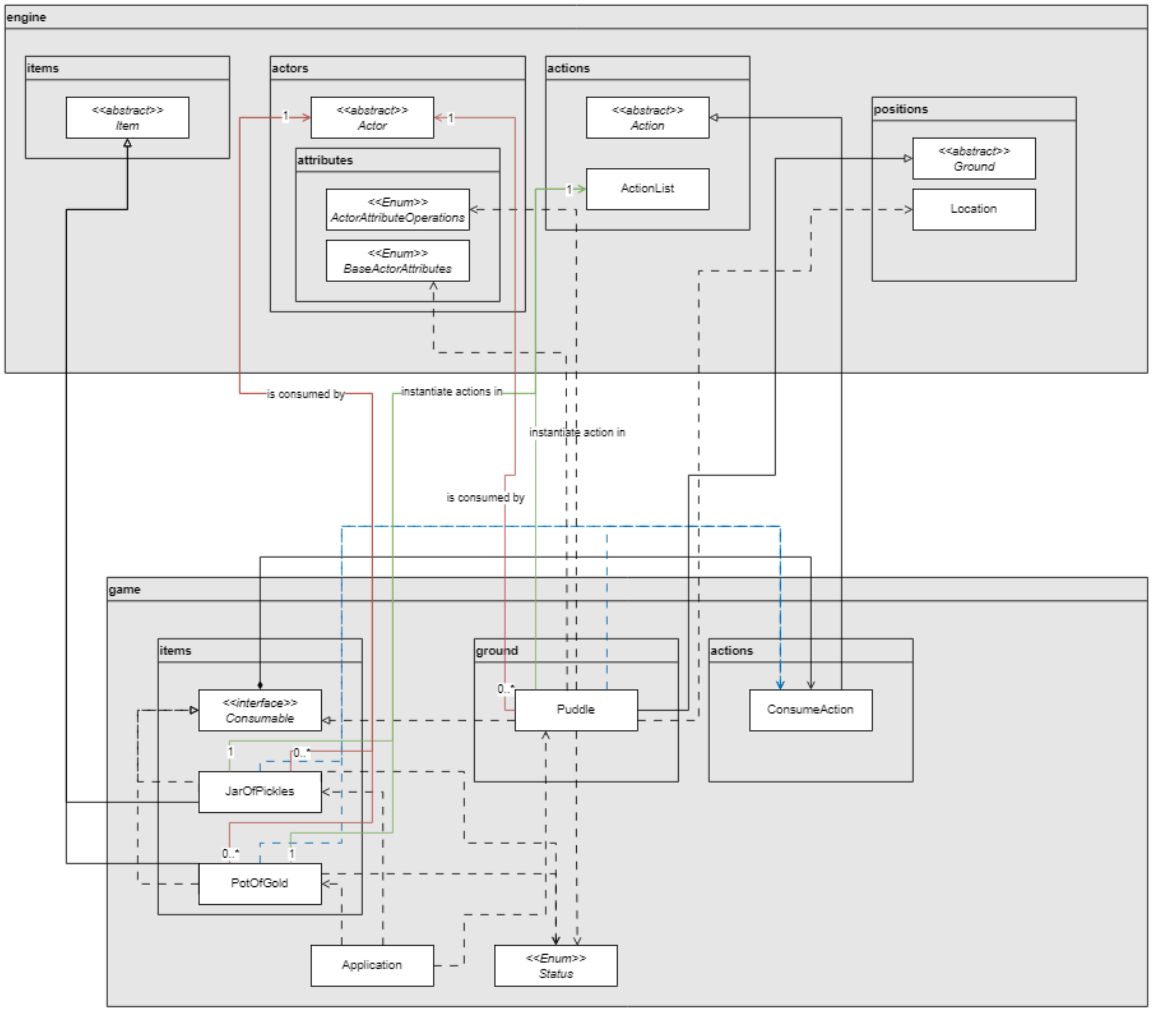
Assignment 2 Design Rationale
REQ3 : More Scraps

UML



Design Goal

The goal of this requirement is to introduce two new items to the game – Jar of Pickles and Pot of Gold. Now, Puddle should also be drinkable by the Player. The Player should be able to interact with these items. Each item should offer unique effects when interacting with the Player.

- Jar of Pickles should be able to heal or hurt the Player when consumed based on its expiry date
- Pot of Gold should be able to add the Player's balance when picked up
- Puddle should be able to increase Player's Max HP when consumed

The goal is to implement these into the existing code, while adhering to the SOLID and DRY principles. Adhering to these are considered as a good design practice as it supports code extendability.

Design Decision

Two classes were created to represent the Jar of Pickles and Pot of Gold. These classes each extend Item and implements Consumable. For each item, the unique Item details were constructed. Puddle class was updated so that it can be consumed by the Player.

Jar of Pickles implements consumable because the Player is able to use it up. For now, the only action that the Player can act upon is consuming it. To implement this, the allowableAction from Item parent class is overridden to instantiate ConsumeAction. Following the requirement, when consumed, it has a 50-50 possibility of being either safe to consume or expired. A boolean value is randomly generated to determine this. Based on the result, the Jar of Pickle will either heal or hurt the Player. This is handled in the consume method, inherited from the Consumable interface.

Pot of Gold implements consumable because the Player is able to use it up. For now, the only action that the Player can act upon is consuming it. To implement this, the allowableAction from Item parent class is overridden to instantiate ConsumeAction. Following the requirement, when consumed, it will add the Player's wallet balance.This is handled in the consume method, inherited from the Consumable interface.

Puddle implements consumable because the Player is able to use it up. For now, the only action that the Player can act upon is consuming it. To implement this, the allowableAction from Item parent class is overridden to instantiate ConsumeAction. Following the requirement, when consumed, it can add the Max HP of the Player. This is handled in the consume method, inherited from the Consumable interface.

Alternative Design

Consumable could have stayed an abstract class, extending an Item and Ground. The Jar of Pickles, Pot of Gold, and Puddle could extend this abstract class, and get its own implementation of the consume method and effects, specific to its functionality.

This alternative implementation could shorten the code for Jar of Pickles and Pot of Gold, as it will only need to extend from the ConsumableItem abstract class. It can inherit the methods from its superclass.

However, the chosen design is preferred as it allows it to be more usable in different contexts. The current implementation allows anything – Ground, Item, or maybe even other Actors – to be a consumable. If the alternative were to be chosen, it "hard-codes" consumables to be of

an instance of item. If in the future other things, such as actors or new entities in the game were to be a consumable, then, other consumable classes have to be made to accommodate this.

Analysis of Alternative Design

The alternative design is not ideal as it violates this principle

Don't repeat Yourself (DRY)

If there are other entities, like Ground, that are consumable, a new Consumable abstract class needs to be created extending that specific entity. That class would essentially include the same functionalities and methods as the original ConsumableItem abstract class. This violates the DRY principle, as it results in redundant code.

Final Design

The implementation follows the following principles

Single Responsibility Principle (SRP)

Each item class (JarOfPickles, PotOfGold, and Puddle) has only one responsibility – to define its respective attributes and effects of the respective consumable item.

For instance, JarOfPickle class is responsible for defining its attributes (name, display character, and portability) and restoring Player's health point when consumed.

Open/Closed Principle (OCP)

The design allows for easy extension – new consumable items can easily be added in without needing to change any of the existing code. For example, if a new type of consumable item (BowlOfRice) were to be added in, the only thing that needs to be done is to create a new class extending item and implementing consumable. In that class, all the appropriate attributes of the BowlOfRice can be handled. This shows that each consumable item class is open for extension but closed for modification.

Liskov Substitution Principle (OCP)

The LSP states that the objects of the superclass can be replaceable with the object of its subclasses, when it is expected. This implementation adheres to this principle because by defining a Consumable interface and allowing the items PotOfGold, JarOfPickles, and Puddle to implement it, when the ConsumeAction is called, and has to be acted upon a consumable, the instances of these objects can be used without causing any errors.

Don't repeat Yourself (DRY)

The implementation follows the DRY principle as it avoids duplicates by encapsulating common behaviour/attributes, such as the consume effect (from Consumable interface) and allowable actions to be performed on it (from abstract Item class).

Each consumable item reuses the common functionality from the superclasses, thus, adhering to the DRY principle.

Pros & Cons

Pros

One of the advantages of this approach is that it supports code modularity. Each consumable item is defined by its own unique attributes, therefore, making it easy to maintain. It is also very extensible – allowing easy addition of new consumable items by extending Item and implementing Consumable.

Cons

On the other hand, this implementation for this requirement results in minor code duplication. There are similarities in the structure of the allowable actions in the Jar of Pickles and Pot of Gold. Both of the allowableAction methods in these two classes checks if a Player is standing at where its placed, and serves an option for the Player to consume it, which is stored in the intialised Actions List.

There is also a scalability issue to consider, wherein if in the future, the number of consumable items keeps on increasing, then additional utility classes and/or layers would be more beneficial to handle the effects accordingly.

Conclusion

The implementation overall follows the SOLID principle, mainly SRP and OCP, effectively minimising duplicates through the DRY principle. While there are room for improvement, the expected outcome to introduce and manage consumable items in the game was achieved.