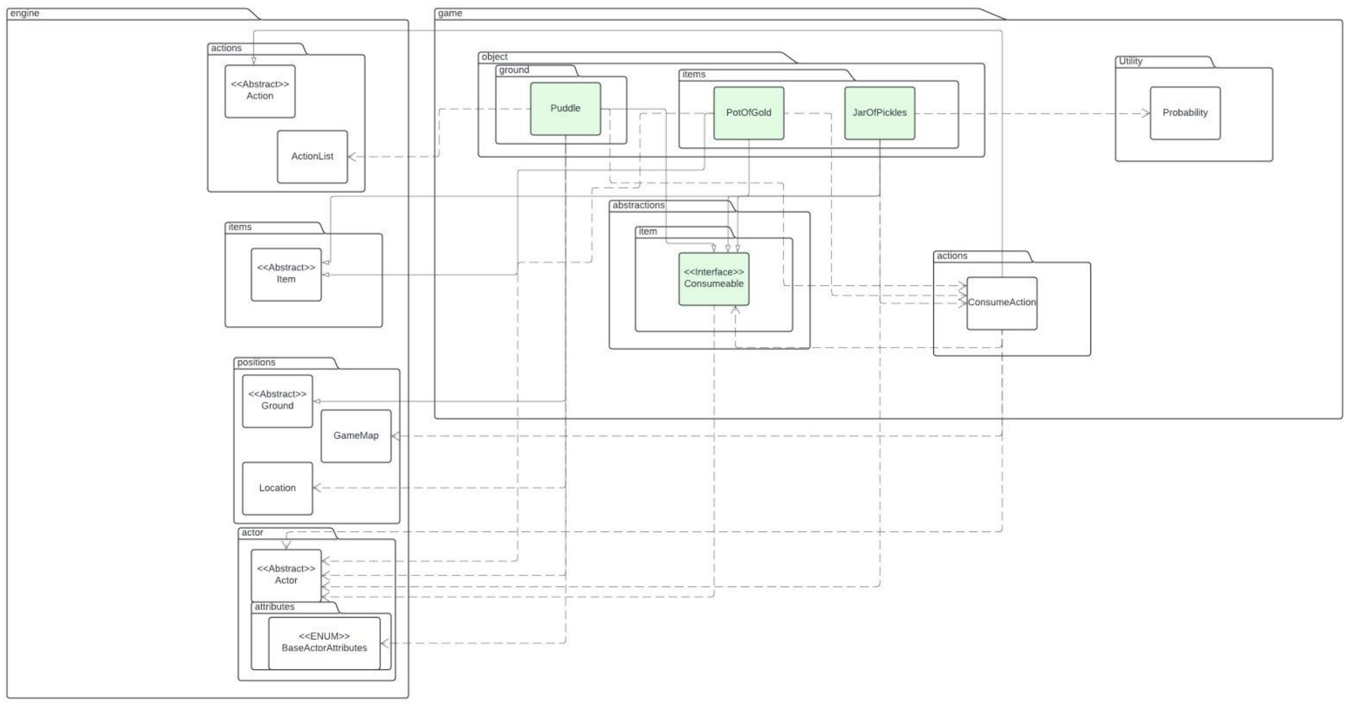
Requirement 3 Design Rationale

UML



Design Goal

The goal of this requirement is to create 2 new items that can be consumed, the Jar of Pickles and the Pot of Gold and extend the Puddle such that it can infinitely and permanently add health to the player. The Pot of Gold once consumed will add 10 credits to the player's balance whereas the jar of pickles will either hurt or heal the actor by 1 point with a 50% chance. Ideally this implementation will have close adherence to SOLID and DRY principles, as well as considerations to future extensibilities.

Design Decision

In implementing the requirement for the player to consume the pot of gold and jar of pickles, the decision was made to define these as distinct classes, each inheriting from a common Item class provided by the game engine and extend the consumable interface which was created by me. The puddle followed a similar implementation however instead of extending the item class it extended the ground class as the puddle is a part of the map and cannot be picked up or moved. All 3 of these classes have the ConsumeAction added to their action list as the player can interact with these objects by consuming them. Each class defines it own consume method however as each object should be responsible for managing what happens to it when it is consumed this does result in repeated code however the benefits gained by the extensibility outweigh the repetition of code.

Alternative Design:

Initially there was no interface consumable it was instead an abstract class called consumable item that extended the item class. This method worked fine for the jar of pickles and the pot of gold as they both managed their own consumption by having their own consume method and did not violate and SOLID principals, however this approach presented challenges when it came to the puddle. Since puddle extended the ground, it could not extend consumableItem and use the ConsumeAction either, puddle was unable to extend item as well since it is part of the ground. This resulted in having to create a separate action called the AddHealthAction and have it added to the puddles allowable actions. Evidence of this implementation can be found in the git logs. Example of the AddHealthAction code is below.

```
package game.action;

import edu.monash.fit2099.engine.actions.Action;
import edu.monash.fit2099.engine.positions.GameMap;
import
edu.monash.fit2099.engine.actors.attributes.ActorAttributeOperations;
import edu.monash.fit2099.engine.actors.attributes.BaseActorAttributes;
import edu.monash.fit2099.engine.actors.Actor;




public class AddHealthAction extends Action {
    private int healthIncrease;

    /**
     * Constructor.
     *
     * @param healthIncrease the amount to increase health by
     */
    public AddHealthAction(int healthIncrease) {
        this.healthIncrease = healthIncrease;
    }

    /**
     * Permanantly adds 1 health point to the actors max health
     *
     * @param actor the actor performing the action
     * @param map   the game map where the action is taking place
     * @return a message describing the result of the action
     */
    @Override
    public String execute(Actor actor, GameMap map) {
        actor.modifyAttributeMaximum(BaseActorAttributes.HEALTH,
ActorAttributeOperations.INCREASE,this.healthIncrease);
        return actor + " Health increased by " + this.healthIncrease + "!";
    }


    /**
```

```
     * Provides a description of the action for displaying in menus or
logs.
     *
     * @param actor the actor performing the action
     * @return a description of the action
     */
    @Override
    public String menuDescription(Actor actor) {
        return "increase " + actor + " health by " + this.healthIncrease;
    }
}
```

This design was not ideal due to the following reasons below:

1. Single Responsibility Principal (SRP)
    a. SRP was violated as the puddle is not handling its own responsibility of being consumed instead that responsibility is handed off to another class, the AddHealthAction class. Therefore, this principal was violated with this approach.
2. Open-Close Principal (OCP)
    a. OCP was also violated as if in the future another ground class was able to be consumed that did had an effect on the player other than permanently adding health another action would have to be created and as more ground classes had this ability the more actions would have to be created and it was not flexible enough.
3. Liskov Substitution Principal (LSP)
    a. This method did adhered to this principle as either of the classes could be substituted where its parent class was required and the game would still be functional.

   For the violations mentioned above this design was not ideal and instead the decision was made to change the abstract ConsumeableItem class to an interface such that classes other than items can be consumable, and every class would manage its own consumption by having a consume method in it. Another approach was to have an abstract ConsumableGround class but that would also require a new ConsumeAction class too which would increase the complexity of the system by having 4 very similar classes and there is still repeated code.

Our Design

Below is the following discussion of how our design adheres to the SOLID and DRY principles.

1. Single Responsibility Principle (SRP):

- **Application**: Each class is responsible for handling a single responsibility: its own consumption
- for modification (existing classes remain unchanged).
- **Why**: This approach allows different classes to have different behaviours when getting consumed
- **Pros**: Very flexible in what happens when an object is consumed

- **Cons**: Lots of repeated code for example if multiple classes heal by 1 amount then all of them will have the exact same code for healing by 1.

2. Open-Close Principle (OCP):
   - **Application**: The system is open for extension (new classes that can be consumed can be added or existing classes can be changed to be consumed) but closed for modification (existing classes remain unchanged).
   - **Why**: This approach allows any class to become a class that can be consumed with minimal impact on existing code, promoting extensibility. This is done by just having the class implement the interface consumeable and have the ConsumeAction added to its allowable actions.
   - **Pros**: Easy to make classes consumable
   - **Cons**: Having repeated code is again the issue as every class that is consumable will have the code adding the ConsumeAction to its allowable actions and each consume action will be implemented. With the abstract class approach we would not have to add the ConsumeAction into the allowable actions.
3. Liskov Substitution Principle (LSP):
   - **Application**: The Jar of pickles, pot of gold and puddle all maintain the functionality of their respective parent classes and can be interchanged without causing issues
   - **Why**: This is crucial for maintaining integrity of the game as all child classes should have the expected behaviour of their parent classes
   - **Pros**: Easy to introduce new item types by extending the Item class or ground class and maintains their functionality.
   - **Cons**: May require additional design considerations if new items substantially differ from
4. Interface Segregation Principle (ISP):
   - **Application**: The consumable interface has minimal responsibility and only handles making an object consumable by having just the consume method in it.
   - **Why**: With this approach an item can do multiple things alongside being consumable which increases flexibility and extensibility.
   - **Pros**: Classes can be spawnable, consumable among other things and extend a base class such as item, ground or weapon.
   - **Cons**: have to implement a unique consume method for each class even if the functionality in that method is not unique.
5. Don’t Repeat Yourself (DRY) Principle:
   - **Application**: For this requirement there is repeated code as that was the cost for adherence to the SOLID principles.
   - **Why**: Since every class must implement its own consume method basic functionality such as heal is repeated among classes that heal
   - **Pros**: Not having repeated code is good because functionality is not repeated and if one change is made somewhere you may not have to change it in many places
   - **Cons**: Can lead to classes becoming large

The chosen design adheres well to the SOLID and DRY principles, providing a robust framework for managing classes that are able to be consumed. Each class is designed with a single responsibility, extending from a common base and implementing an interface to implement the consumable functionality, which allows for clear and maintainable code structure. This approach not only ensures that enhancements can be made with minimal impact on existing functionalities but also sets a strong foundation for future requirements and extensions.