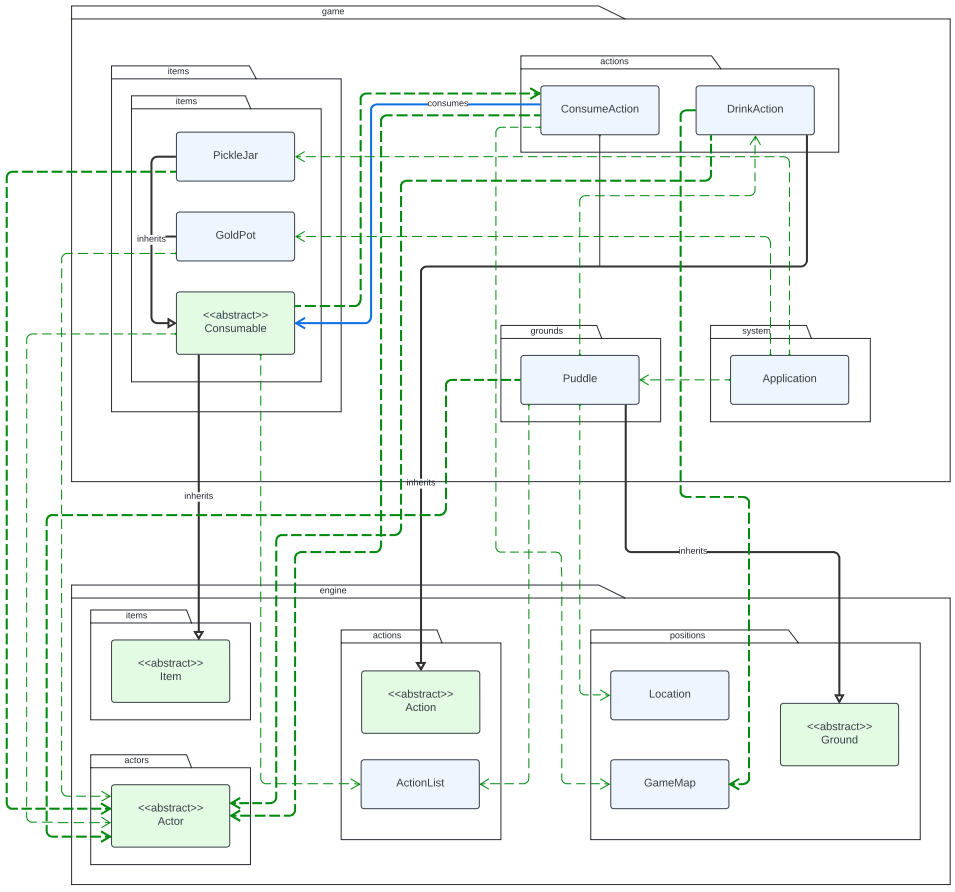
FIT2099 Assignment 2 Design Rationale
Requirement 3: MORE SCRAPS

UML Diagram:



Design Goal:
The design goal for this requirement is to extend the gameplay experience by allowing a Player to consume various items to increase attribute points such as health or wallet balance while emphasizing clean code practices such as simplicity, abstraction, and modularity of code. The objective is to integrate these new elements such as interactive consumables and environmental features seamlessly with the existing game structure, promoting maintainability and extensibility by adhering to best coding practices.

Design Decision:
In implementing the consumable items and environments such as “Jar of Pickles”, “Pot of Gold” and “Puddle of Water”, the decision was made to utilise and extend the existing classes within the engine (e.g. Item, Ground) to create further abstract classes to implement each concrete class. Specifically, the decision to create the abstract class “Consumable” which is then used to extend concrete classes such as “PickleJar” and “GoldPot” which instantiate objects to be used in the application. This design allows for

abstraction of these concrete classes, as their consumable functionality does not need to be known by the end-user, as well as extensibility, as future consumable classes can easily be implemented using the Consumable abstract class. This approach maintains consistency with the game's coding standards and structure and promotes polymorphism and inheritance to introduce new features without redundant code, enhancing the system's modularity and readability.

Alternative Design:
One alternative approach could involve creating separate, specialised classes for each type of consumable item and a distinct system for environmental interactions. However, this would lead to a significant decrease in the maintainability, readability and extensibility of code. A small change in one class may snowball and lead to many changes being required throughout the system to reflect a small difference.

Analysis of Alternative Design:
The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Single-Responsibility Principle**
    - This would violate the SRP because implementing separate systems for each new feature would lead to classes with multiple responsibilities e.g. the "PickleJar" class would be responsible for constructing a PickleJar object, as well as the consumption functionality and the after-effect functionality.
2. **Open-Closed Principle**
    - This would violate the OCP as significant modifications to existing code would be required to accommodate new features or inclusions with the code, which contradicts the principle of extending rather than modifying existing code.
3. **Don't Repeat Yourself Principle**
    - This approach would violate the DRY principle as code would have to be duplicated across multiple classes that have similar functionalities, as there would not be a parent class that each concrete class could extend from. This approach would lead to classes with similar functionalities having duplicated code with very little changes.

Final Design:
In our design, we closely adhere to SOLID principles to ensure code quality and future extensibility:

1. **Single-Responsibility Principle**
    - **Application:** Each class is tasked with only one responsibility and only has one clear and specific role. For example, the "PickleJar" class only handles behaviours exclusive to the "PickleJar" concept, such as expiry, hurt/heal points. It does not need to be aware of the implementation of consume, which is handled by another class.
    - **Why:** This allows for the abstraction of code, so some classes do not need to know specific implementations of certain methods. It promotes extensibility, as further classes can be created by extending a parent class and does not require modification of other classes. It also promotes maintainability of code.
    - **Pros:**
        - Improves maintainability
        - Requires careful design to avoid overlap class responsibility
    - **Cons:**
        - May result in many classes being created if they don't share exact features
        - The need to specify each different method for each class

2. **Open-Closed Principle**
    - **Application:** New features and additional classes are implemented through extensions of parent classes and implementation of interfaces, which allows for the existing code to remain unchanged. For example, the abstract class “Consumable” extends from “Item” class and additional consumable items extend from the “Consumable” class. This is done without modification of any parent class.
    - **Why:** Enhances flexibility of the system and makes the process quicker, easier and simpler when accommodating any changes in the future. Promotes extensibility within the code.
    - **Pros:**
        - Improves flexibility and extensibility
        - Makes it easier to include new features
    - **Cons:**
        - May increase complexity of code if not managed well
3. **Liskov-Substitution Principle**
    - **Application:** Objects of a superclass are replaceable with objects of its subclasses without affecting the correctness of the program. LSP is shown in the use of the “Consumable” interface and its subclasses “PickleJar” and “GoldPot”. In the application, anywhere a “Consumable” object is expected, it can be substituted with an object of one of its subclasses, as shown in the “ConsumeAction” class.
    - **Why:** Allows for the easy and seamless substitution of different types of consumables without needing to change the existing code.
    - **Pros:**
        - Renders the system more modular and extensible
        - Simplifies maintenance
    - **Cons:**
        - Can lead to overgeneralisation of code if not careful

Conclusion:
Overall, our chosen design provides a robust framework for enhancing gameplay while adhering to the best practices in coding. By carefully considering the SOLID principles and good design principles such as simplicity, modularity, and abstraction, we have developed a solution that is both efficient and scalable, paving the way for future game enhancements and feature integrations.