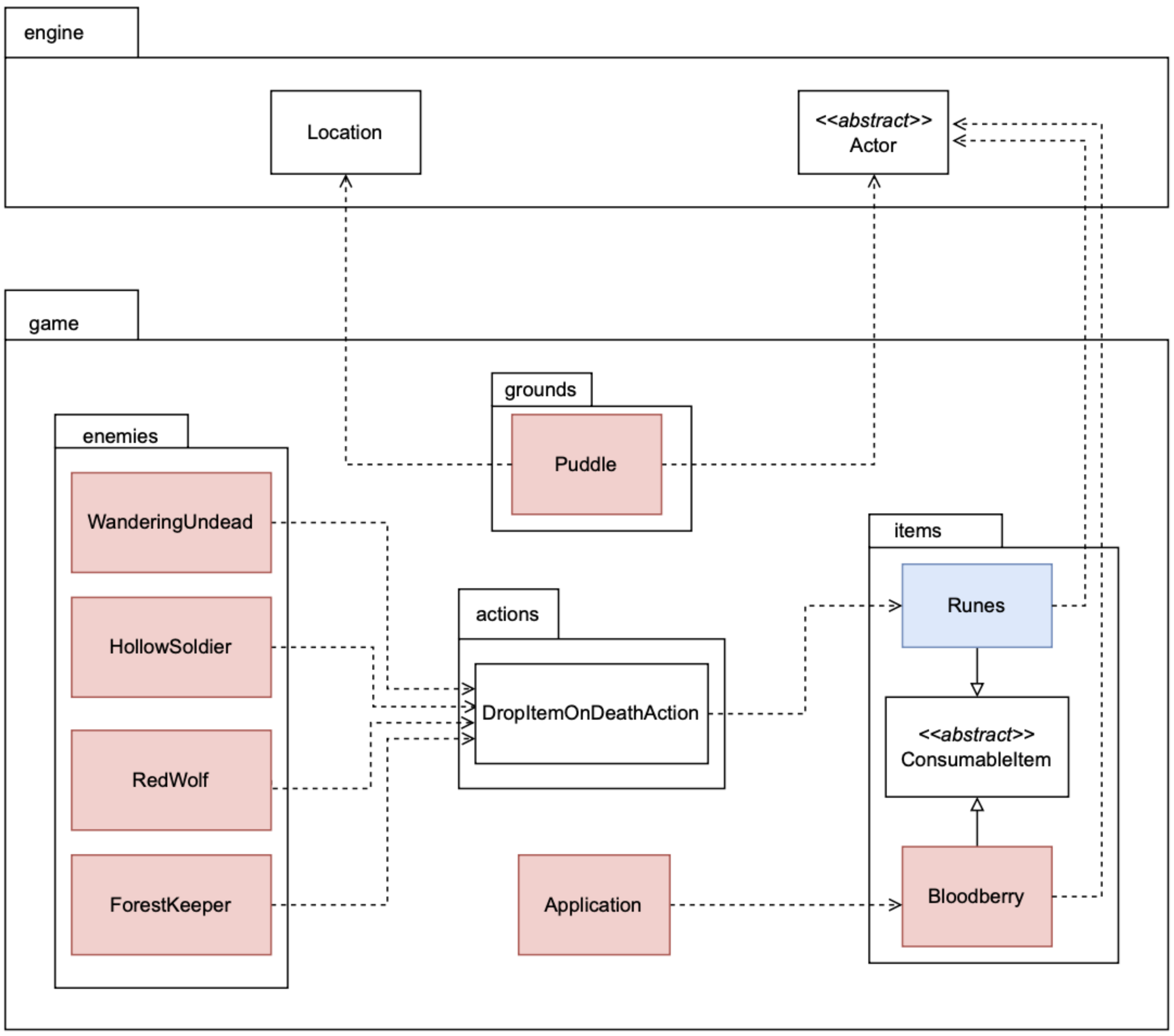
# Design Rationale (REQ 2)



## Changes Made from Assignment 1

ConsumeHealingVialAction and ConsumeRefreshingFlaskAction are removed from the code, replaced by a single ComsumeItemAction class. A ConsumableItem abstract class is added and is used for all the consumable items.

By refactoring the comsumption actions of items like HealingVial and RefreshingFlask into a unified ConsumableItem abstract class and a corresponding ConsumeItemAction, the design follows the principles of SOLID and DRY. Firstly, the Single Responsibility Principle (SRP) is exhibited as the responsibility of item consumption is decoupled from the specific type of item and centralized within the ConsumableItem and ConsumeItemAction classes, ensuring that any changes related to consumption logic are localized.

The Open-Closed Principle (OCP) is also upheld as new consumable items can be introduced without modifying existing consumption action logic, simply by extending the ConsumableItem class. The DRY principle - Don't Repeat Yourself, is clearly represented as the code for consuming items is centralized and not duplicated across different item-specific action classes. This design thus improves the scalability and maintainability of the codebase.

## Implementation

### Classes Added

1. Runes extends ConsumableItem
2. DrinkAction extends Action
3. Bloodberry extends ConsumableItem

### Codes Modified

1. A value attribute has been assigned to Runes to ensure that every instance of Runes is accompanied by a value.
2. The consume method from the parent class (ConsumableItem) has been overridden to allow the user's wallet balance to be increased by the Runes' value.
3. DrinkAction has been added to the Puddle's allowableActions method:
4. Whenever the player steps directly on the Puddle, player's health and stamina will automatically be increased every tick.
5. The consume method in the Bloodberry class has been modified to permanently increase the player's maximum health by 5 points.
6. The player's playTurn method has been modified to display the wallet balance in the console every game round.
7. The unconscious method for all enemies has been modified so that Runes are dropped upon their death (Add new DropItemOnDeathAction).

### Abstract Class ConsumableItem

This abstract class was created to represent any consumable item.

**Design Principles Followed**

1. **Single Responsibility Principle (SOLID)**
   - The Consumable abstract class primarily handles the functionalities related to a consumable item, such as effects of consuming the item and creation/addition of a ConsumeItemAction.
2. **Open-Closed Principle (SOLID)**
   - The consumable item class adds a new layer of abstraction to represent all items that are consumable. This class contains all functionalities that a consumable item should have (consume method, allowableAction method). This means any extensions of the system involving the addition of new types of consumable item can be achieved by the addition of new classes that extend the consumable item class without a need to modify the this class.
3. **Liskov Substitution Principle (SOLID)**
   - The subclasses (Runes, RefreashingFlask, etc.) are substitutable for their base class (ConsumableItem). This is evident in methods like allowableActions where the subclasses call the superclass's method via super.allowableActions() and then add their unique behaviours. The consume method is implemented in the ConsumableItem class as an abstract method for its child classes to overridee according to its functionality.
4. **DRY (Don't Repeat Yourself)**

   - The common functionalities and properties among the consumable items such as the creation of ConsumeItemAction are kept in the ConsumableItem class, ensuring there's no repetition. Specific functionalities, such as item-specific events upon consumption is implemented within the child classes.
5. **High Cohesion**
   - The ConsumableItem class focuses solely on functionalities related to consumable items. This design ensures that related functionalities are bundled together in a logical manner.

### Runes and Bloodberry

The classes Runes and Bloodberry represent specific types of consumable item within the game.

#### Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
   - The Runes and Bloodberry classes represent specific item types in the game. The Runes class encapsulates the value attribute that represents the amount of currency to be added, and performs the adding of balance to the player's wallet via the consume method, and the Bloodberry class encapsulates the effect of consuming a bloodberry to an actor.
2. **Open-Closed Principle (SOLID)**
   - These classes are open for extension (can be subclassed to create variations) but closed for modification. Their core behaviors can be extended by overriding methods, but the core structure remains unchanged. For example, when a new currency (type of Rune) is introduced, no modifications are needed in the Runes class to represent its variantions. Similarly addition of variants of the bloodberry does not require addition of attributes within the Bloodberry class.
3. **Liskov Substitution Principle (SOLID)**
   - Since both Runes and Bloodberry extend the ConsumableItem class, it is expected to be substitutable for ConsumableItem. Both Runes and Bloodberry overrides the abstract method consume from its super class ConsumableItem.
4. **High Cohesion**
   - High cohesion is maintained in these classes. The method and properties within this class revolve around a single concept: addition of an actor's attribute when upon consumption.

#### Pros of the Design

1. **Abstraction-Extensibility**
   - Abstract class ConsumableItem was created instead of an Consumable interface to allow its child classes to return the specific action (ConsumeItemAction) from the allowableActions method. Addition of this new layer of abstraction not only makes the system more comprehensible, but also improves extensibility of the system as any items that are consumable can extend this custom abstract class.

#### Cons of the Design

1. **Complexity**
   - Although this new layer abstraction successfully represents a group of items, it adds a new complexity in abstration layer within the system, limiting the future extension of the system. For example, this might limit addition of new abstract classes of the Item class as the child classes can only extend from one parent class.