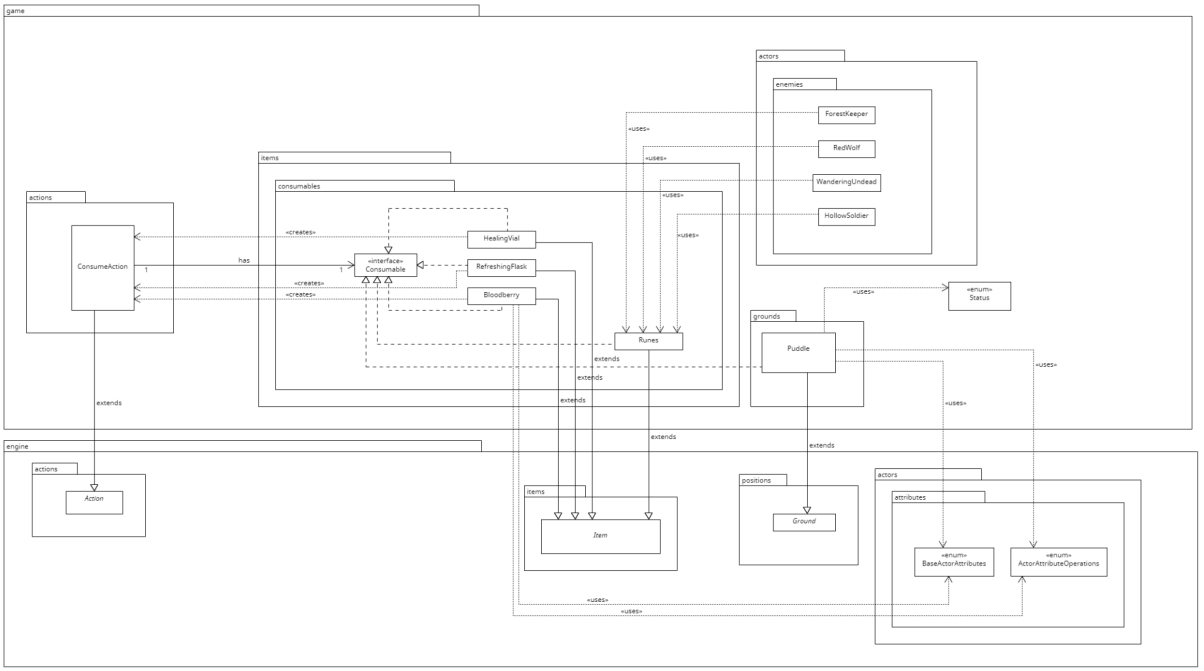

Changes from Assignment 1

We decided to change how consumable items are used in general. In assignment 1 we had all items that can be consumed extend an abstract ‘ConsumableItem’ class, which implements the ‘Consumable’ interface. We realised that the abstract ‘ConsumableItem’ class just adds another layer of abstraction without much benefit, as we can just do all the modifications to the actor’s HP, Stamina etc. through the ‘consume’ method of the interface. This adheres to the Interface Segregation Principle (ISP), as we created a specific ‘Consumable’ interface that contains only the methods relevant to the implementing classes.

Now, all items that can be consumed directly implement the ‘Consumable’ interface. ConsumeAction now takes in a ‘Consumable’ and has a ‘Consumable’ as its attribute, we call the Consumable’s ‘consume’ method inside the execute. This adheres to the Dependency Inversion Principle (DIP), as we rely on the abstraction of the ‘Consumable’ rather than its specific implementation.

HealingVial, RefreshingFlask, Bloodberry and Runes can be used interchangeably in contexts where an ‘Consumable’ is expected, like when passing the argument through the ‘ConsumeAction’ constructor, this demonstrates adherence to the Liskov Substitution Principle (LSP).

With the changes mentioned above, we allow for more extensibility for future consumable entities, not just items, but grounds like Puddles as well. This adheres to the Open/Closed Principle (OCP), as we can easily add more entities that can be consumed by the player without modification to the existing code.

The main objective for REQ2 are to add Runes, that are dropped by enemies upon their death, add ‘Bloodberry’ item that permanently increases the max HP of the player, and lastly, allow the player to drink from Puddles, which increases the Stamina of the player.

‘Runes’ implements the ‘Consumable’ interface, all enemies drop Runes inside their ‘unconscious’ method. With our previous implementation of using an abstract ‘ConsumableItem’ class, we could not add the functionality to consume from a puddle easily, but now we can just have Puddle implement the ‘Consumable’ interface. Lastly, Bloodberry also implements the ‘Consumable’ interface.

Advantages over the old implementation:

Extensibility

Our implementation is easily extensible for future consumables that might be added. Future consumable entities can just implement the 'Consumable' interface without requiring modifications to existing code, this adheres to the Open/Closed Principle (OCP).

Simplified Hierarchy

Removing the abstract 'ConsumableItem' class simplifies the class hierarchy. There's no need for an extra layer of abstraction, which makes the codebase cleaner and easier to understand. This aligns with the Single Responsibility Principle (SRP) by ensuring that each class has a clear and specific purpose.

Disadvantages:

Refactoring Effort

Implementing these changes required quite a bit of refactoring of the existing codebase to remove the abstract 'ConsumableItem' class and update the 'consume' logic in consuming actions.

Testing Complexity

Our implementation required additional testing to ensure that all implementations of the 'Consumable' interface worked correctly, which also took time.