Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 2: Deeper into the woods</u>

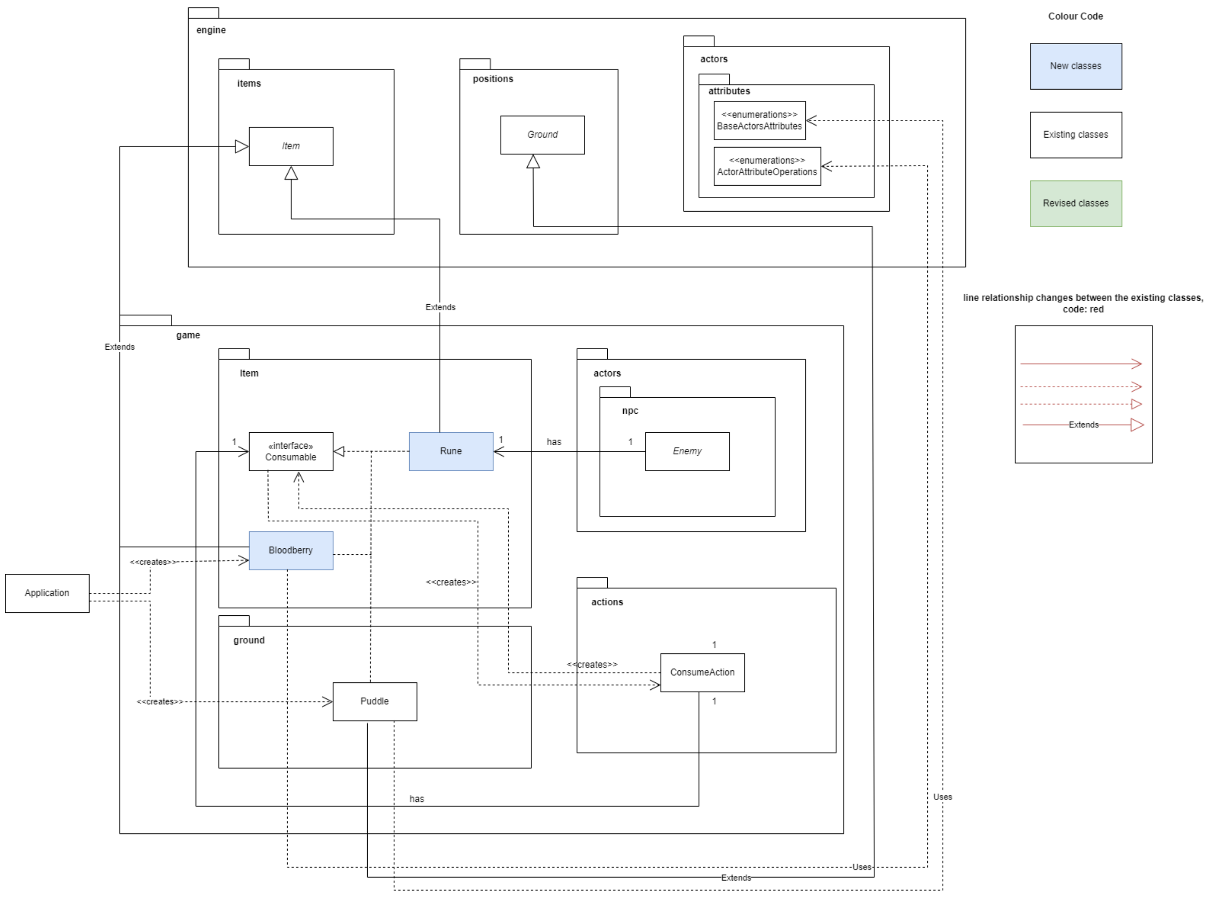

Figure 1: UML class diagram of REQ2: Deeper into the woods

In requirement 2, the “Rune”, “Bloodberry”, and the revised “Puddle” classes are created to follow the Single Responsibility Principle of the SOLID principle, which is only responsible for consumption. For example, “Rune” class focuses on adding the balance of the player, “Bloodberry” class focuses on increasing the maximum health of the player, “Puddle” class focuses on healing and increasing stamina of the player.

“Rune” and “Bloodberry” extend the “Item” class to perform polymorphism (“Puddle” extends “Ground” class), and hence prevent code smells in the design. Besides, these three classes implement the interface “Consumable”, this aligns with the Open/Closed Principle as these classes must utilise the “consume” method in the “Consumable” interface, while the extension for the “consume” method is allowable. This means that these classes might have different implementations with the “consume” method. Also, with OCP, it ensures the extensibility of the programs, for example, if a new item/ground which can be consumed by the player is added, it can implement the “Consumable” interface.

The implementation of the “Consumable” interface and extension of the “Item” abstract class also aligns with the Liskov Substitution Principle because the “Rune” “Bloodberry”, and “Puddle” classes can perform the same functionalities expected from the abstraction class and interface. Moreover, the “Consumable” interface is small enough (only contains “consume” method) for the implementation of these classes, it does not violate the Interface Segregation Principle.

With the “Rune” class passed into the constructor of the “Enemy” class, dependency injection is applied. When the subclasses of the “Enemy” class require a “Rune” object, it can be directly created in the argument of the subclasses’s constructor with the value of the “Rune”. For instance: “super("Forest Keeper", '8', 125,new Rune(50));”, where “50” is the value of the “Rune”, this value can be modified according to different subclasses of the “Enemy”. This helps the design follow the dependency inversion principle.

However, if the abstractions (“Consumable” and “Item”) experience any changes, the subclasses will be affected and they might need to implement the changes even if some are unnecessary. This would introduce connascence which will lead to more chances of bugs.