Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 5: The Inhabitants of The Burial Ground

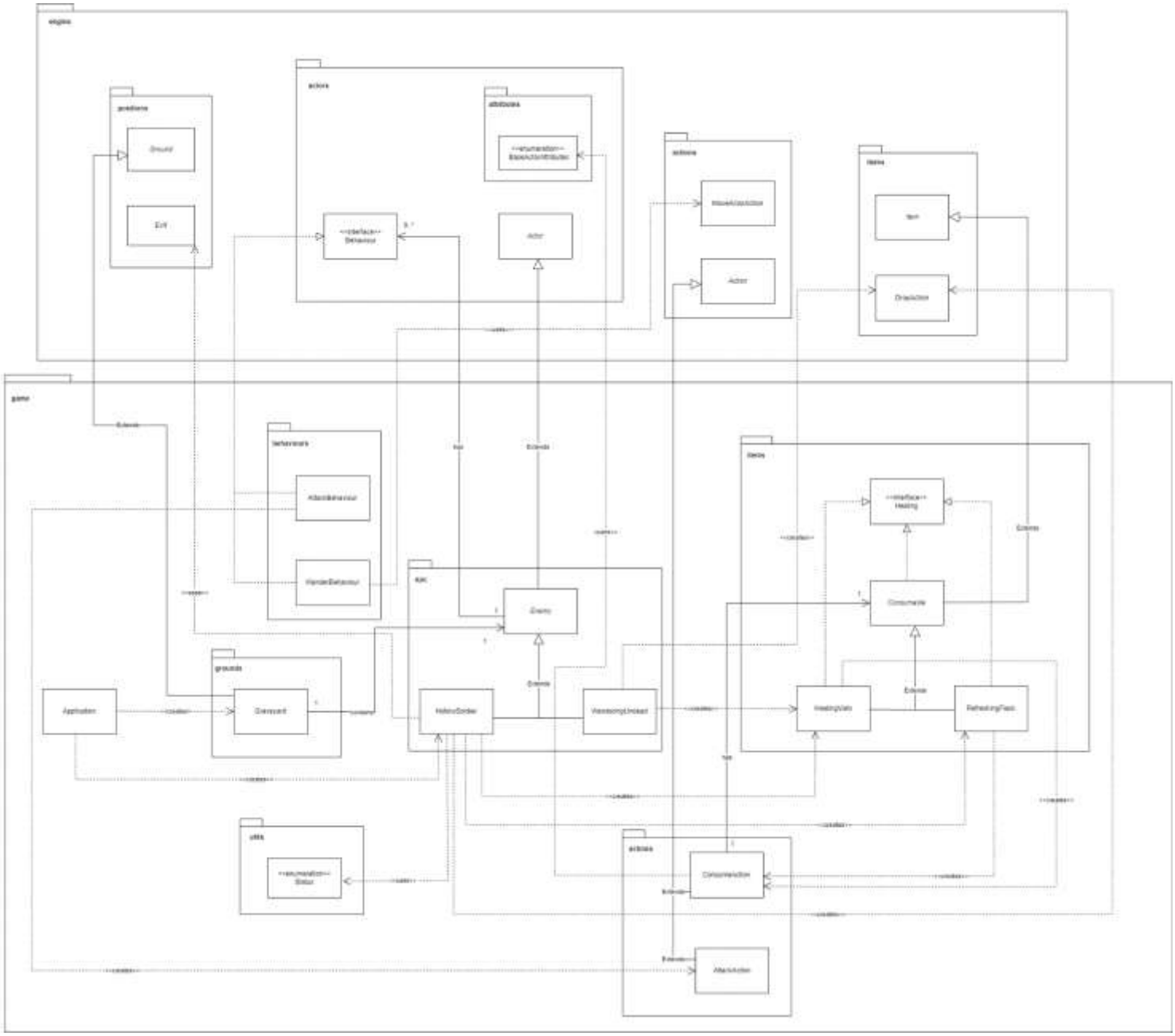

Figure 1: UML class diagram of REQ5 - The Inhabitants of The Burial Ground

The provided diagram shows the entire system implementation for Requirement 5, The Inhabitants of The Burial Ground. The primary goal of this design is that the Graveyard in the Burial Ground to spawn new types of enemies, which is Hollow Soldier at every turn each turn of the game, creating healing vial and refreshing flask items which will be dropped by an enemy once it is defeated. Thus, there are new classes are introduced in the extended system, HollowSoldier class, Healing interface class, HealingVials class, RefreshingFlask class, ConsumeAction class and an abstract Consumable class. Those new created classes will interact with the existing classes in the system.

The Application class creates (dependency relationship) Graveyard class in different game map, The Burial Ground, as remark on in previous design rationale REQ4: The Burial Ground. The Application class also creates (dependency relationship) instance of HollowSoldier class. As stated in the previous design rationale REQ2, the Graveyard class is able to spawn different enemies at certain chances since it has association relationship with the abstract Enemy class. Thus, the Graveyard class will spawn new types of enemies, which is HollowSolider.

The new enemy class added into the extended system is HollowSolider class that extends to the abstract Enemy class. This HollowSolider class same as the WanderingUndead class that proposed in the previous design rationale, have a dependency relationship with the Status enum class to check every actor who it encounters before performing any actions to them. Therefore, the HollowSolider can prevent to attack their own kind and only attack other actors including the player. Alternatively, if the Status enum class does not be used, we will need to implement if-else statement and instanceof operator (checking class type) in AttackBehaviour class to ensure that enemies of the same type will not attack each other which involves extra dependencies and violates the Open/Closed Principle (OCP) because when new enemies are added, the existing code of the AttackBehaviour class have to be modified to ensure the condition of enemies not attacking their own type holds true at all times. Besides, the HollowSolider class also has dependency with Exit class to determine its surrounding contains any actors. This can help this enemy to decide its behaviour to be performed towards that actor. The abstract Enemy class will also have an association relationship with the Behaviour interface class. In addition, the HollowSoldier class also creates (dependency relationship) DropAction to drop refreshing flask and healing vials once it is defeated.

Moreover, the Healing interface class is introduced which will be implemented by an abstract Consumable class, HealingVials class and RefreshingFlask class. We decided to make the Healing as an interface class as it stores collection of abstract methods for healing or buffing to be implemented in the Item class in later implementations which has ability to buff or heal the holder of the item. Having Healing as an interface class so that classes are introduced in future implementations which have ability to buff or heal the owner can implement it. This approach not only reduces code repetitions (DRY), but it also easily to maintain and extend our system if any new features are introduced in later implementation. We also introduced an abstract Consumable class extends the abstract Item class from the engine and implements the Healing interface class. Consumable class is an abstract class that representing all items that can be consumed. The main reason for having this design decision is that there are various consumable items will exist in the World, and some items have the ability to heal or buff the Actor, while some may not. Thus, any new Consumable subclasses are introduced, they may not necessarily have to implement the abstract methods except for classes that have ability to heal or buff the owner of the item. Thus, classes that does not implement interface methods that offered in the Healing interface class will not violate the Liskov Substitution Principle (LSP) as they still perform same functionality as its base class. Besides, we have designed the RefreshingFlask class and HealingVials class extend from Consumable abstract class as well as to

implement Healing interface class since both share common traits, which both are consumable items and also have special ability to heal or buff the player.

However, the design decision of the above paragraph also comes with several issues. The main issue of the design decision is that the system will become more complex if more classes that have special abilities are introduced in the later implementation of the game. Besides, our system is hard to maintain as every class that implements the Healing interface class must implement all the interface methods in the interface class. Some classes have special abilities, but some methods offered in the interface class may not suit those classes. This may cause us to refactor the code, which results in violating the Open/Closed Principle (OCP). To solve this problem, we can introduce new interface classes so that new classes have those abilities just implements that interfaces classes that they require. This approach will help our design to align with Interface Segregation Principle (ISP).

To let our owner to consume an item, we proposed a new class, ConsumeAction class extends from the abstract Action class. The ConsumeAction has an association relationship with the abstract Consumable class so every Consumable subclass that pass into the ConsumeAction constructor can be differentiated and execute different method before passing back result to the engine. This can prevent any code smells such as downcasting as polymorphism can be used. The ConsumeAction also has dependency relationship with BaseActorAttributes enum class to show the attributes of the Player after consuming the items that have special abilities for the current implementation. Moreover, RefreshingFlask class and HealingVials class creates (dependency relationship) ConsumeAction so that the Actor who owns instances of both classes can perform consume action on them.