Design Rationale

Name: Koe Rui En

ID: 32839677

<u>REQ 4: The Burial Ground</u>

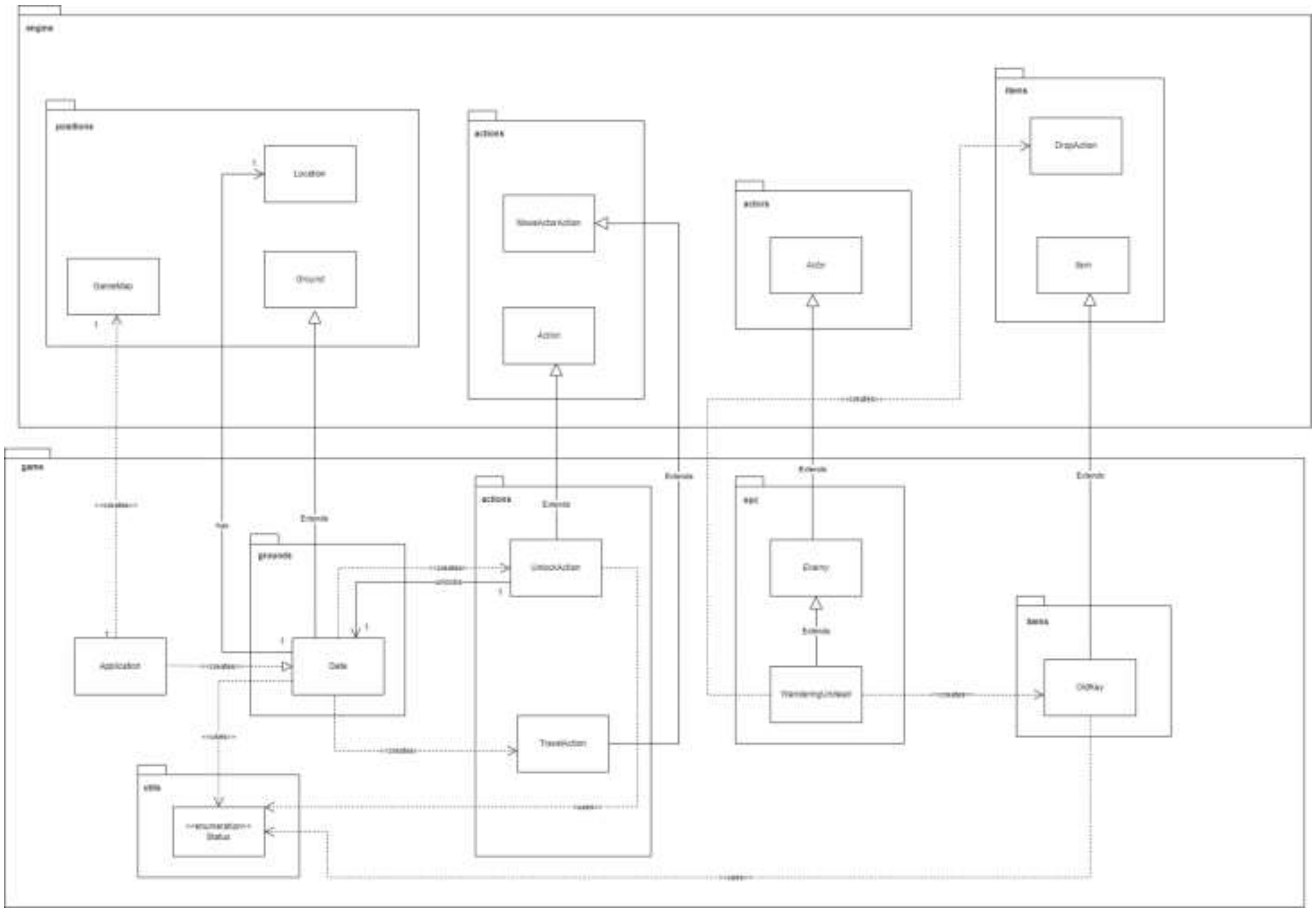

Figure 1: UML class diagram of REQ4 – The Burial Ground

The provided diagram shows the entire system implementation for Requirement 4, The Burial Ground. The primary goal of this design is to create a new game map, which is known as Burial Ground, a gate to travel to the new game map and a key to unlock the gate. Therefore, some new classes are proposed in this system, UnlockAction class, TravelAction class, Gate and OldKey class. The existing classes in the game and engine packages will interact with those classes.

To achieve our design goal, we decided to have OldKey class extends from Item abstract class from the engine since old key is an item. The OldKey class also has a dependency relationship with Status enum class, so when the Gate clas check the Actor whether they have the instance of OldKey class, the Gate class can prevent use of instanceof operator to check the item actual class which to match the OldKey class. This can reduce dependency on OldKey class and prevent the violation of Open/Closed Principle (OCP) if any new Item class is created which may have a dependency relationship with GateClass in future implementation. Thus, the existing code in GateClass to check the item in the Player's inventory to check the existence of an instance of OldKey class does not need to be modified and our design has aligned with Open/Closed Principle (OCP).

Furthermore, we also decided to have the Gate class extends the Ground abstract class. The advantage of having Gate class as ground as it can access the relevant methods that provided by the base class, Ground abstract class. In this circumstance, the Gate class needs to return travel action and unlock action to the engine to be further processed by using allowableActions method, so any Actor can teleport to another location through it. Thus, it has a dependency relationship with TravelAction class and UnlockAction class. It also has an association relationship with the Location class from the engine to teleport the Actor to the destination. By doing so, our system is more easily to be extended and maintained if more instances of Gate classes are created to help the Actor to travel to various locations.

To determine whether the actor can use the gate to teleport to another location, the UnlockAction class needs to check the Actor's inventory has instance of OldKey by using status of instance of OldKey class, so it has a dependency relationship with Status enum class. The Gate class also use Status enum class to check the current Actor whether has qualification to use it to travel to another location as the Actor has its own unique status.

UnlockAction class is a class that extends the abstract Action class from the engine as it handles all the unlock actions in the game. It has an association relationship with Gate class in the current implementation, so it can override the status of gate class if the gate is successfully to be opened as it has the attribute of Gate class. If any classes involve unlocking action in later implementations, those classes can create (dependency relationship) the UnlockAction class and the UnlockActions can also has an association relationship with those classes. The new features can be added in the UnlockAction related with those classes. As such, our design decision to create UnlockAction class has followed the Single Responsibility Principle (SRP) where a class should a class should have only one reason to change, as well as creating reusable code for the future implementations of the game. Thus, our system is easily to be extended. The disadvantage of this design decision is that the code to check the item may be repetitive and breach the Don't Repeat Yourself (DRY) principle if more items to be checked.

The TravelAction class is a class that extends the MoveActorAction class from the engine as it has common functionalities with MoveActorAction class as the TravelAction class is to teleport the Player to their destination. The TravelAction class specifically handles this interaction so that we can know the name of the destination the Player teleports instead of using coordinates of the location or destination. Alternatively, we could just create (dependency relationship) MoveActorAction class.

However, the downside to this is that the MoveActorAction class does not have the attribute of name of the destination to be teleported to be displayed in the console menu. This it may confuse the user who plays a role as the Player. Our design adheres to the Liskov Substitution Principle (LSP) where a subclass can do anything its base class do and it does not disrupt the behaviour of our system.

Besides, the Application class will create a new GameMap class and Gate class, so the Application class has a dependency relationship with the GameMap class and the Gate class. As mentioned in previous design rationale REQ2 and REQ3, the WanderingUndead class extends from the Enemy abstract class. The WanderingUndead class creates (dependency relationship) OldKey class from the game and DropAction class from the engine, so it can perform drop key action once it is defected by the player.