Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 1: The Player

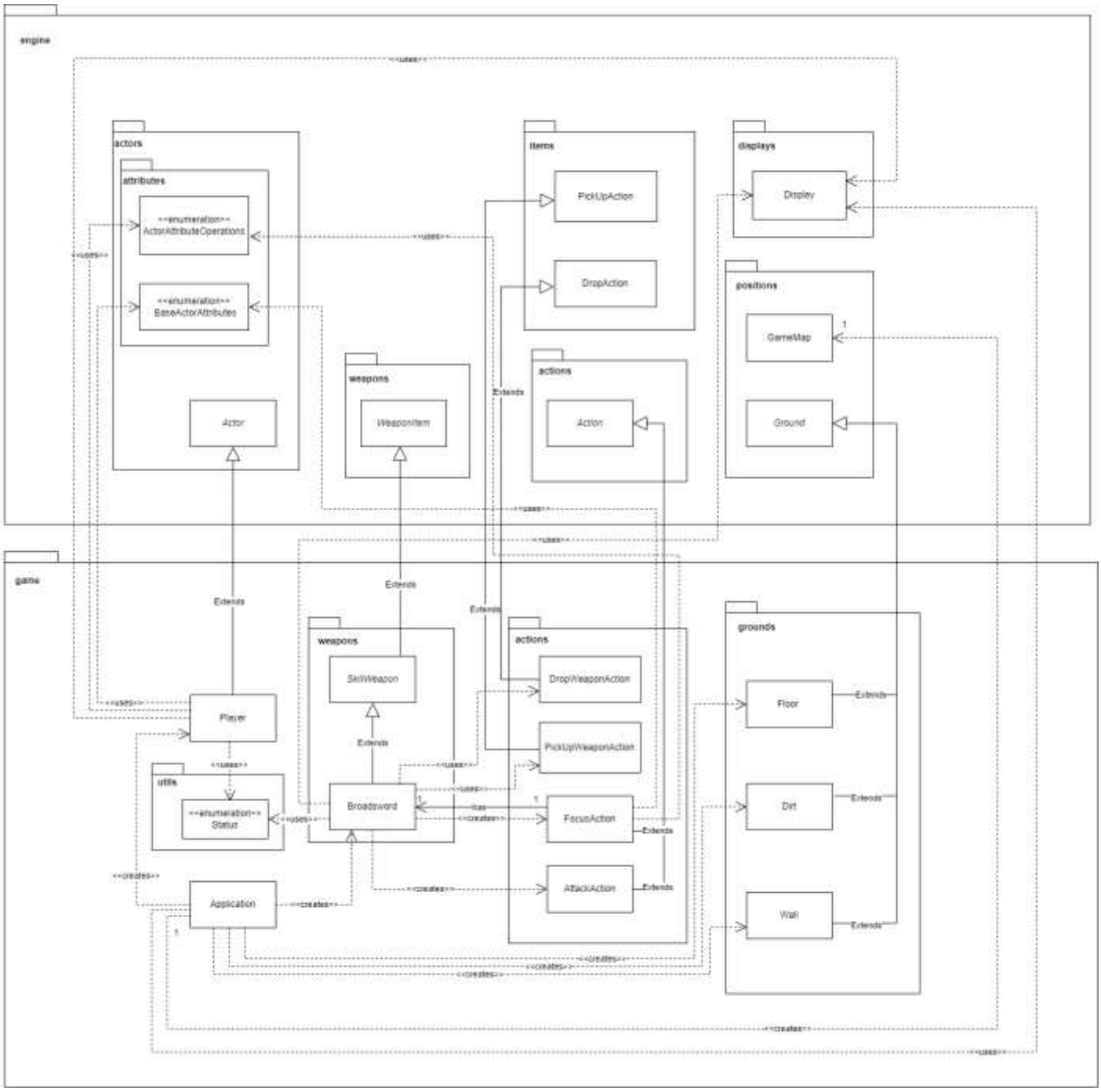

Figure 1: UML class diagram of REQ1 – The Player

The provided diagram shows the entire system implementation for Requirement 1, The Player. The primary goal of this design is that to create the player, “The Abandoned Village” game map, a weapon which have special skill and the building on the game up which is made up dirt, floor and surrounded by wall. In this requirement, several new classes are introduced to the existing system, which is Broadsword class, Skill Weapon class, DropWeaponAction, PickUpWeaponAction class and FocusAction class. New classes will interact with the existing classes in game package as well as classes from the engine.

The existing system includes a Player class, which extends the abstract Actor class from the engine. This Player class will be further extended to incorporate additional functionality. In this requirement, the Player class has dependency with Display class to display their details at every turn of the game. This Player class also uses BaseActorAttributes enum class and ActorAttributeOperations enum class to modify their attribute, which is stamina at every turn of game in the current implementation. The The Player class will be assigned a status to signify their status, thus resulting in dependency relationship towards this Status enum class. It is noted that status assigned to the Player class will be a generalised and flexible status. The advantage of having Status enum class is to differentiate the Actor, so classes are introduced in the later implementation that involves any status checking can use it before performing certain tasks or actions, and this may avoid violation of Open/Closed Principle (OCP).

Furthermore, we also decided to have SkillWeapon class as an abstract class and extends the WeaponItem abstract class from the engine. The reason we have this design decision is that not all the weapon items will have special skill. This approach can help us to differentiate the weapon item type. Besides, any new implementations made in this SkillWeapon abstract class will apply to other skilled weapon which means prevention of Don’t Repeat Yourself (DRY) as we avoid repeating codes in every weapon class. As we need to create a weapon for the player to attack the enemy, The Broadsword class is introduced. Since the Broadsword has special skill, Focus, so this Broadsword class extends the abstract SkillWeapon class. This Broadsword class also has a dependency relationship with Status enum class, so its status can also be used in the future implementation and if the Player who owns it will also have extra status, so we can identify the Actor more easily that have different types of status in any new implementation of the game. We also introduced the FocusAction class since this FocusAction class is a special skill of the Broadsword and extend the Action abstract class from the game engine. Thus, the Broadsword class has a dependency to the FocusAction class. The FocusAction class has an association relationship with the Broadsword class as it need to modify the damage multiplier and hit rate of instance of Broadsword class. Since the player also use the Broadsword to attack other actors, so this Broadsword class creates (dependency relationship) with the AttackAction class that extends from the abstract Action class.

We also introduced new 2 classes, PickUpWeaponAction class and DropWeaponAction class. PickUpWeaponAction class extend from PickUpAction class from the engine, while the DropWeaponAction extends from DropAction class from the engine. The reason we proposed these classes as mentioned in the requirement, the status of skilled weapon after being dropped and picked up again. By doing so, we can reuse these classes for the skilled weapons regardless skill types whenever those weapons are dropped or picked up again by the Actor. Besides, separating the implementations of the PickUpWeaponAction and DropWeaponAction of the skilled weapon to different classes, our design decision has aligned with the Single Responsibility Principle (SRP). Thus, the Broadsword class also has a dependency with these PickUpWeaponAction class and DropWeaponAction class since the Broadsword is a skilled weapon.

Moreover, the Application class creates (dependency relationship) Floor class, Dirt class and Wall class as well as GameMap class and Player class. This class also has dependency on Display class to display the message on the console menu.

On the other hand, our design may increase the complexity of the system as many new classes are introduced and increase the difficulty for the developers if any errors are occurring during development.

Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 2: The Abandoned Village's Surroundings

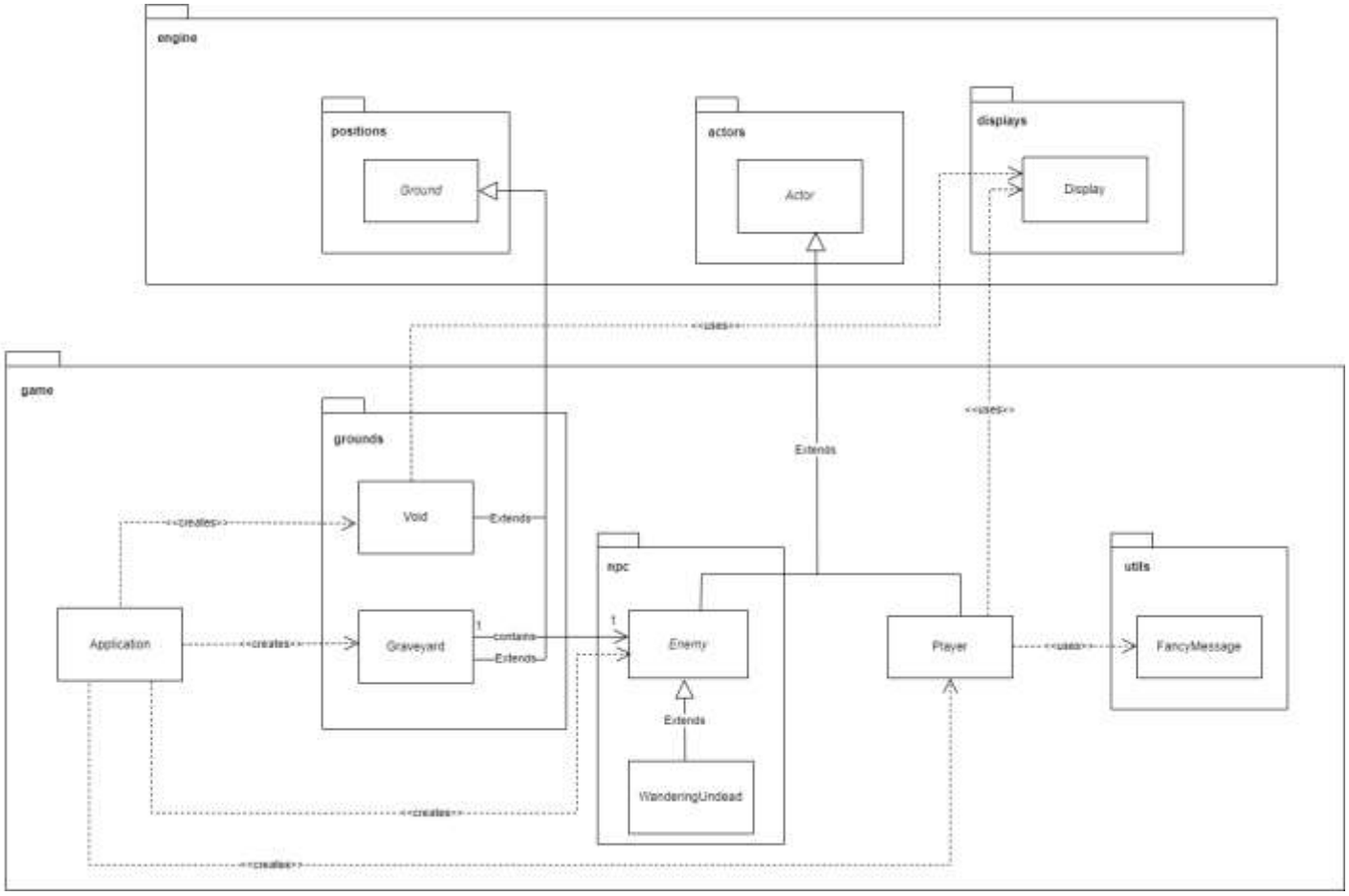

Figure 1: UML of REQ2 - The Abandoned Village's Surroundings

The provided diagram depicts the entire system implementation for Requirement 2, which involves the Abandoned Village's Surroundings. The primary objective of this design is to introduce new classes, specifically the Void class to eliminate all entities that step on it and the Graveyard class to spawn enemies during each turn of the game. In this extended system, three new concrete classes (Void class, Graveyard class, and WanderingUndead class) have been introduced, along with one abstract class called Enemy.

We decided to have both the Void class and Graveyard class extend the abstract class Ground from the engine to avoid redundancy and aligns with the Don't Repeat Yourself (DRY) principle. This decision stems from the fact that these classes share common attributes and methods. Additionally, the Graveyard class has an association relationship with the abstract Enemy class to spawn different enemies at certain chances during each turn of the game. This design choice reduces multiple dependencies (ReD) on various classes that may be implemented in the future, such as the WanderingUndead class. Importantly, this approach adheres to the Liskov Substitution Principle (LSP) since it doesn't violate LSP by using the instanceof operator or object.getClass().getName() to identify the actual subclass before spawning enemies.

The advantage of treating both Void class and Graveyard class as types of ground is that neither of them needs to move, and both can access relevant methods from the abstract Ground class. They can override these methods to implement different implementations. For example, the Void class can access the tick method to track whether an actor steps on it and immediately eliminate the actor, while the Graveyard class can implement the tick method to spawn enemies during every turn of the game, regardless of the player's location. This design aligns with the Open/Closed Principle (OCP) since Void class and Graveyard class extend the functionality of the ground without modifying existing code. However, a potential disadvantage is that code maintenance might become challenging as the system grows larger and more complex.

Furthermore, we have designed the Enemy classes as an abstract class that extends the abstract class Actor from the engine. All inhabitants or enemies in the game will extend this Enemy abstract class because they share common attributes and methods. Enemies in the current implementation, Wandering Undead will extend this Enemy abstract class. This approach ensures that when additional requirements are introduced in future game implementations that apply to all enemies, these new features can be added to the abstract Enemy class, preventing code repetition (DRY). Nevertheless, a drawback is that all subclasses extending from the Enemy abstract class must implement any abstract methods defined in the class, potentially leading to unnecessary implementations in those classes.

Additionally, the Void class uses the Display class to show a message indicating the actor's death. The Player class has dependencies on both the Display class and FancyMessage class to print the death message in the console menu. The Application class creates Void class, Player class, Enemy class and Graveyard class, so the Application class has dependency with those classes.

Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 3: The Wandering Undead

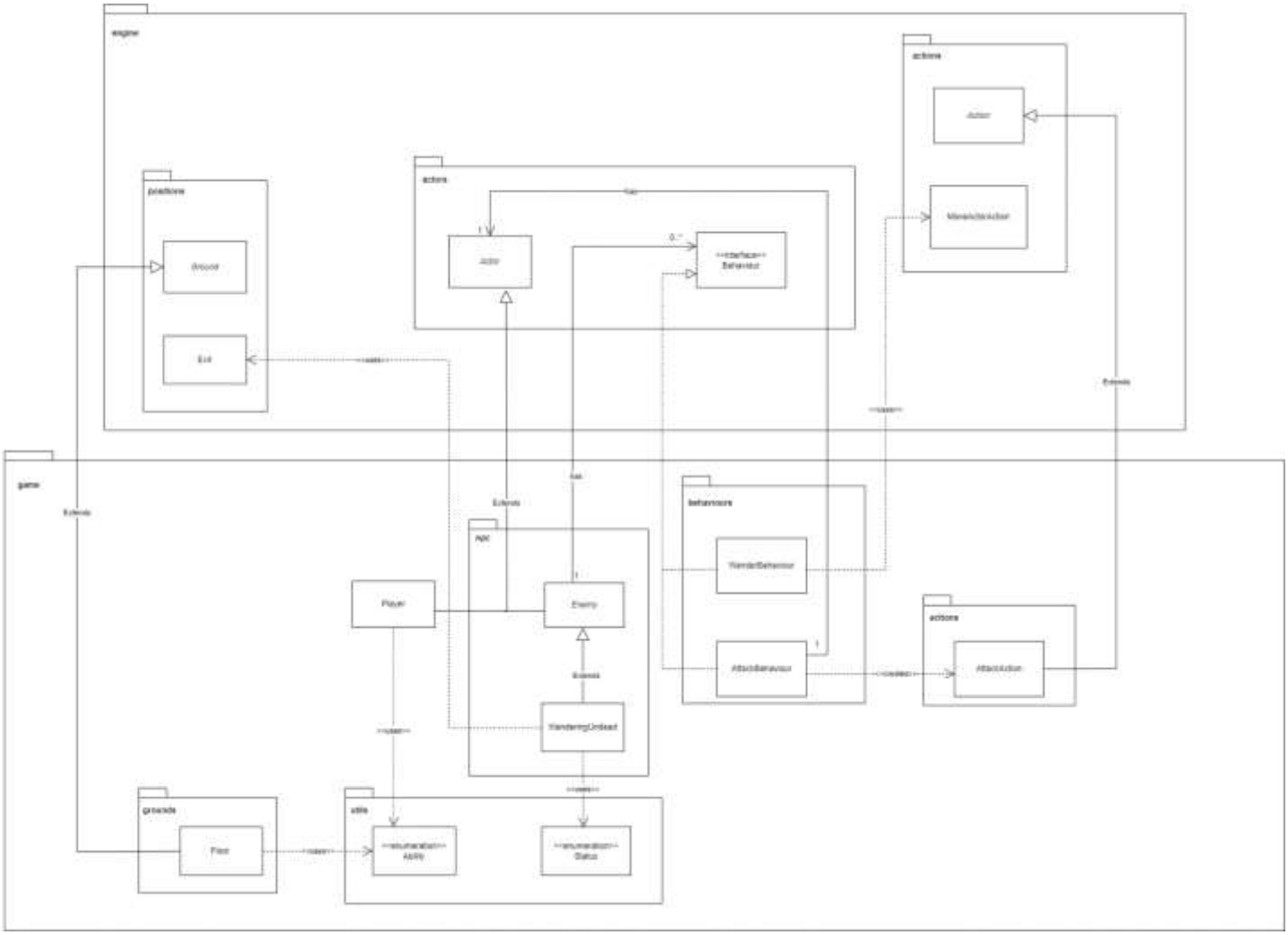

Figure 1: UML class diagram of REQ3 – The Wandering Undead

The provided diagram shows the entire system implementation for Requirement 3, The Wandering Undead. The primary goal of this design is to implement the first enemy that the player encounters which is Wandering Undead who has various behaviours, such as attack and wander and to check the actor so they can enter the floor. There are new classes are proposed in this system, WanderingUndead class, AttackBehaviour. The existing classes in the game package, Player class, Status class, AttackAction class, WanderBehaviour class, Ability class and Floor class, will interact with new classes that mentioned above and also with the existing classes.

The Floor class extends from the abstract Ground class from the engine as well as the Player class extends from the abstract Actor class from the engine, as this was mentioned in the previous design rationale of REQ1: The Player. In this requirement, we decided to have an Ability enum class and Player class will be assigned an ability to do something, thus resulting in a dependency relationship towards this Ability Enum class. Furthermore, the Floor class also has dependency relationship with this Ability enum class to check the current actor who step on it whether they are qualified to pass this terrain. The main reason to have this Ability enum class is to assign different abilities to the player perform various tasks based on their abilities. Alternatively, if the Ability enum class does not be used, we will need to use if-else statement and instanceof operator (checking class type) in Floor class method to check the current Actor actual subclass which involves extra dependencies. Thus, our design adheres to the Open/Closes Principle (OCP) because assigning a flexible ability to the actor so that the existing code in the Floor class does not need to be modified even if more actors will be introduced in the later implementation of the game.

Furthermore, the WanderingUndead class extends from the abstract Enemy class also stated in the previous design rationale of REQ2: The Abandoned Village's Surroundings. In this requirement, we decided to have the WanderingUndead class have a dependency relationship with the Status enum class to check every actor who it encounters before performing any actions to them. Therefore, the Wandering Undead 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 WanderingUndead class 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.

Since every enemy (inhabitant) regardless Wandering Undead share same behaviour, we also decided to have abstract Enemy class have association relationship with the Behaviour interface class which will be implemented by the WanderBehaviour class and the AttackBehaviour class. The AttackBehaviour class will handle any attack actions that are executed by the enemies and not initiated by the player, so this AttackBehaviour class will create AttackAction class to perform the attack action, indicating there is a dependency relationship between AttackBehaviour class and AttackAction class. In addition, the AttackBehaviour is also associated with the Actor abstract class so that every enemy can perform attack actions on that actor. Besides, the WanderBehaviour class will handle the movement of all enemies, so this WanderBehaviour class has a dependency relationship with the MoveActorAction from the engine. Our design is not only can reduce multiple dependencies between enemies and their behaviours, but it also aligns with the Dependency Inversion Principle (DIP) where all of behaviours depend on abstractions instead of concrete implementations. In such

way, we can prevent to perform modifications to our existing code in all our Enemy classes if any new Behaviour class is introduced can be easily extended in later implementation.

Design Rationale

Name: Koe Rui En

ID: 32839677

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

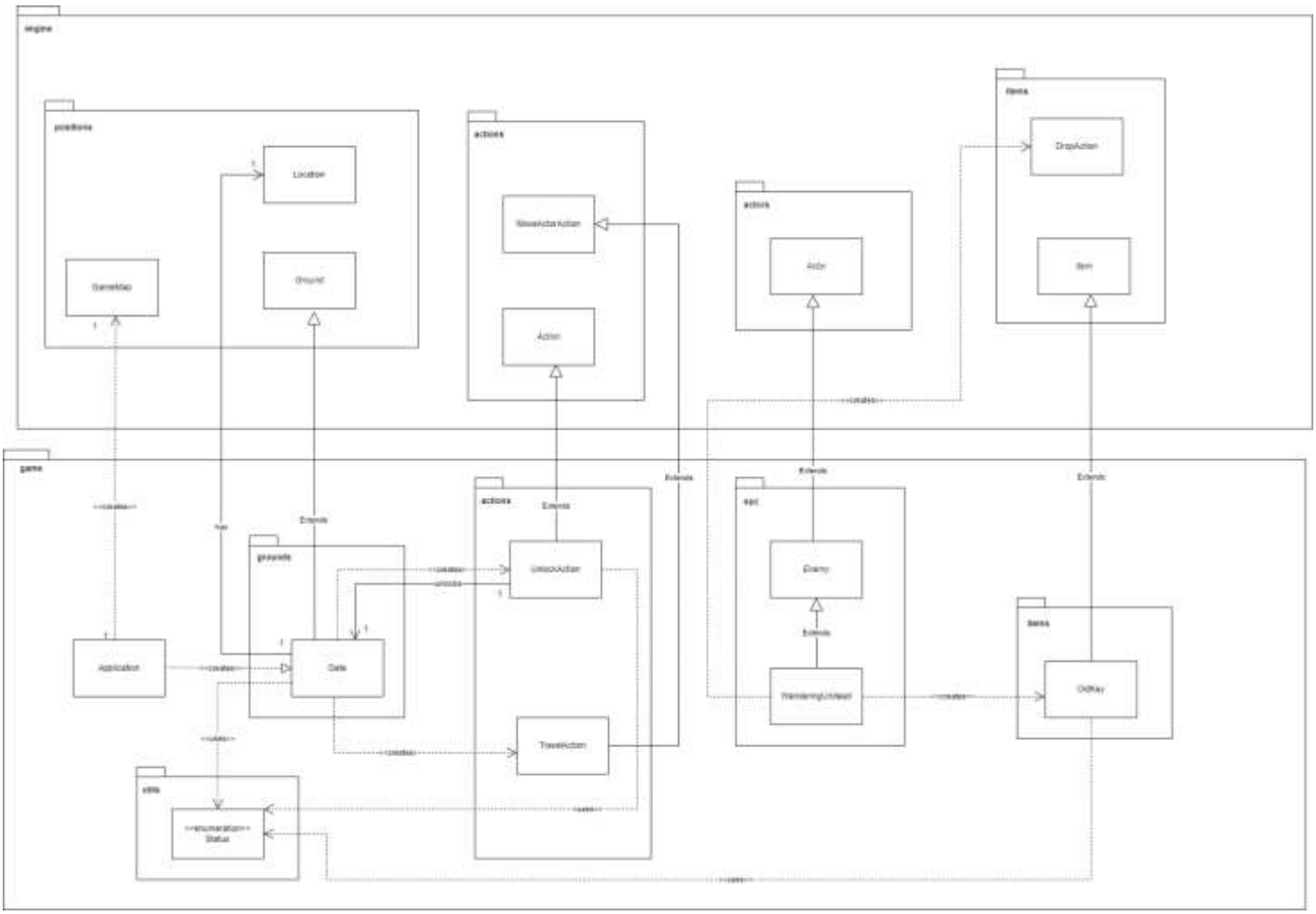

Figure 1: UML class diagram of REQ4 – The Burial Ground

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.

Design Rationale

Name: Koe Rui En

ID: 32839677

REQ 5: The Inhabitants of The Burial Ground

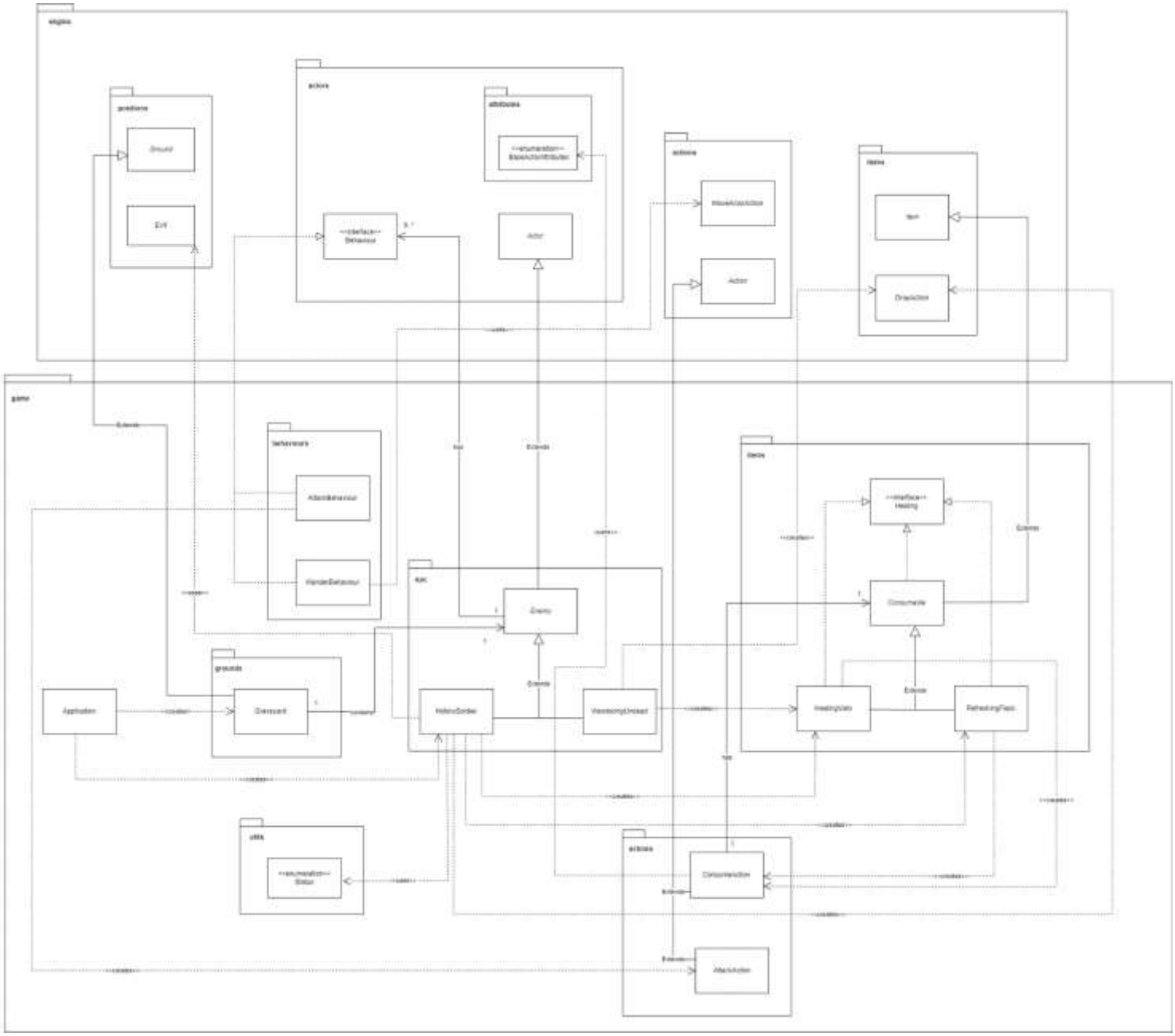

Figure 1: UML class diagram of REQ5 - The Inhabitants of The Burial Ground

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.