Name: Yoong Qian Xin

Student ID: 32846157

FIT2099 Assignment 1: UML Diagram and Design Rationale

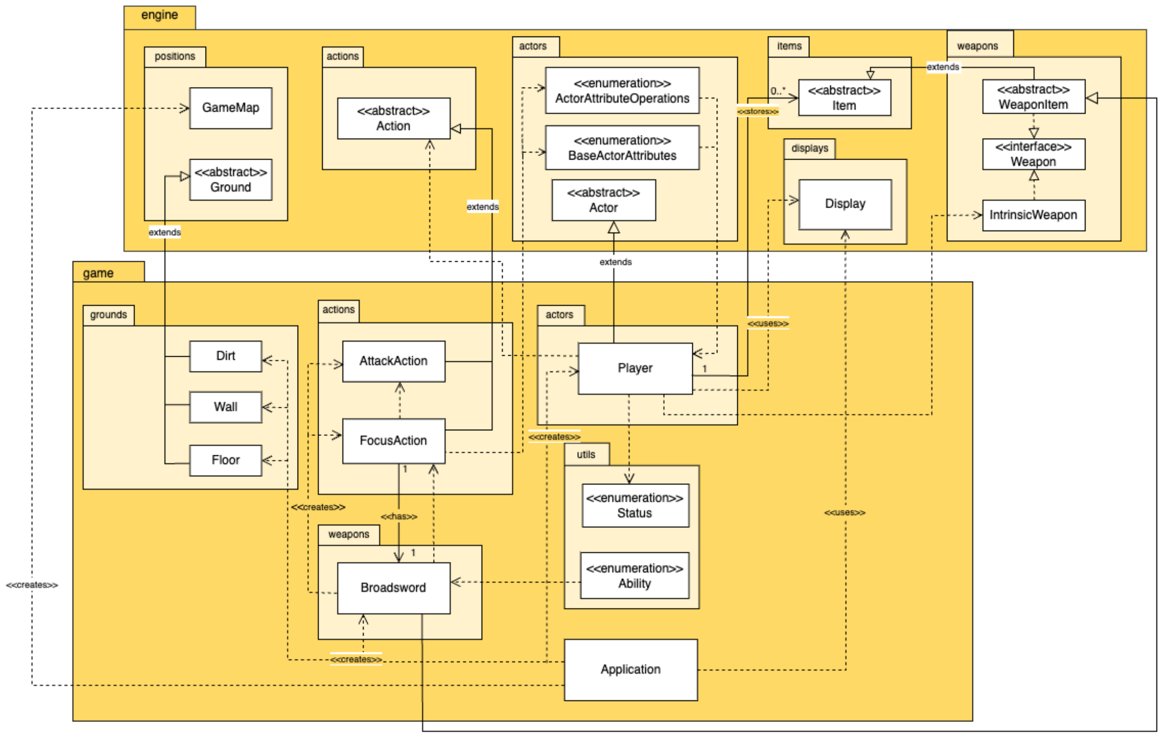
Project Summary

The objective of this project is to develop a text-based game named Designborne. This is a text-based "rogue-like" game and different symbols and alphabetical characters are used to depict various elements within the game world, such as terrain, enemies, weapons. As the project progresses, we introduced new classes that will interact with both the existing classes within the game package and those from the game engine in each subsequent phase. As the project expands in scope, our objective is to consistently adhere to the SOLID principles while pursuing our design goals.

Design Goals

- Design a system that is easy to maintain and modify for the future implementation
- Design a system that adheres to best practices in object-oriented programming

REQ1: The Player



This diagram represents the functionalities stated in requirement 1 of the assignment. In requirement 1, we have to deal with the different terrains, where player starts the game by waking up in the building in the middle of the map. The game map is made up of dirt, floor and surrounded by wall.

We introduced 3 new concrete classes which are Dirt, Floor, Wall, which extends the abstract Ground class from the engine. Also, we created a package named "ground" to store these places. Three of these classes inherit the Ground class because they share common attributes and methods of Ground class. By using inheritance in this way, we can improve the code reusability and reduce the avoid repetitions (DRY), making this design easier to maintain and efficient. In this case, each class will have a dependency relationship with the Application class. as well as GameMap class and Player class. Additionally, this class relies on the Display class to present messages within the console menu.

The existing system currently involves a Player class that extends the abstract Actor class within the engine. This Player class is set to undergo further expansion to include additional functionality. In this case, the Player class has dependency on the Display class to present their information at each game turn. Additionally, the Player class utilizes the BaseActorAttributes enumeration class and the ActorAttributeOperations enumeration class to make adjustments to their attributes, specifically stamina, during each game turn in this requirement. The Player class has dependency on Status enumeration class. The inclusion of the Status enumeration constant serves the purpose of distinguishing the Actor. To illustrate, the Player has a status of HOSTILE_TO_ENEMY. This approach allows for the introduction of classes in subsequent implementations, enabling them to check the status before executing specific tasks or actions. This proactive approach can help prevent any violations of the Open/Closed Principle (OCP).

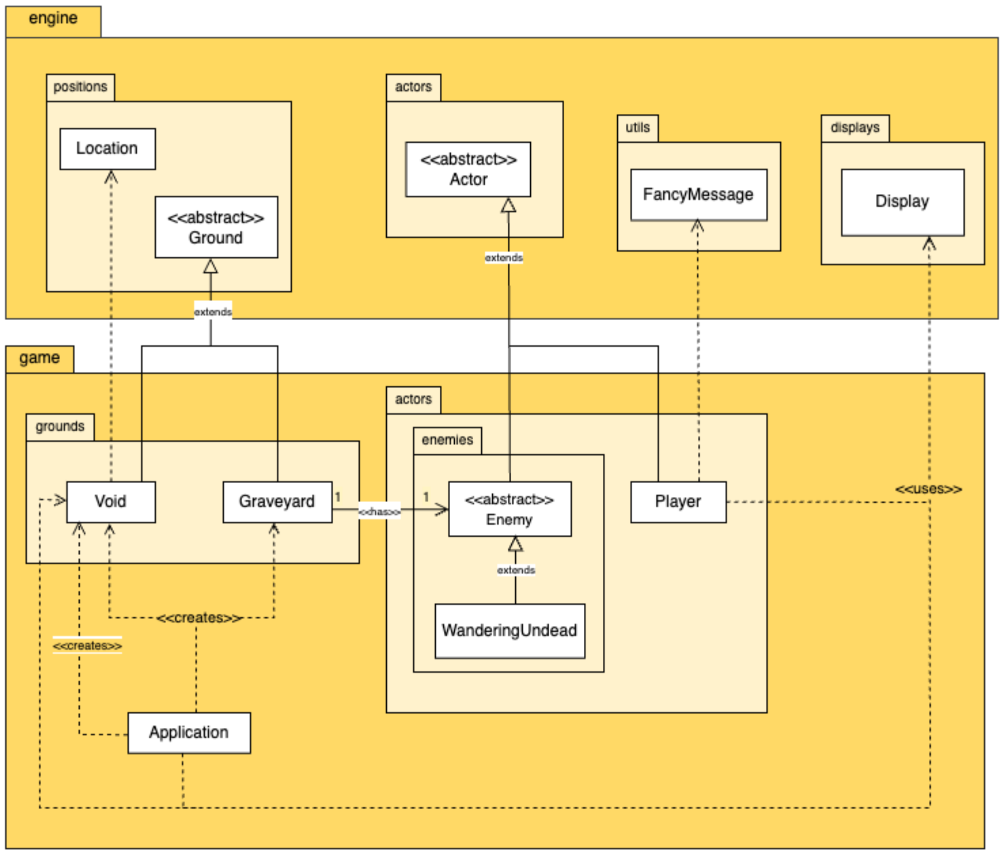
Besides, we introduced 2 new concrete classes, which are Broadsword (added in weapons package), FocusAction(actions package). Broadsword is a weapon which have common attributes and methods with WeaponItem, it is logical to extends the WeaponItem abstract class in the engine provided to avoid repetition (DRY).

In our design, we utilize an enumeration class which is known as Ability in the existing system. We introduced an enumeration constant called WEAPON_SPECIAL_SKILL. The reason why we using this class is because not all the weapons have a special skill, by using the enumeration constant to check whether if the weapon owns a special skill. The advantages of using a new enumeration constant is we don’t need to compare all the weapon item that are provided one by one, we just need to compare the type of the special skill as we already initialize the type of the special skill action in the respective weapon item class when attacking. This can make our game's logic simpler, the code less complex, and it will be easier to maintain and expand in the future. If there are new weapon item that need to be added into the program, we can utilize the enumeration constant to check which special skill action needs to be executed. Therefore, Ability class will have a dependency with Broadsword class.

Even though there is only one weapon item and one special skill stated in the requirement, we decided to separate the weapon’s special skill by creating FocusAction instead of SpecialAction because the special skill of each weapon is not the same (SRP). If we do not separate the special actions, SpecialAction class will have too much responsibility and we have to check the type of weapon before actually executing the special weapon skill. As such, Broadsword has dependency on FocusAction because it needs to return a FocusAction instance when the system asks it for allowable actions.

FocusAction has association with Broadsword as it needs to know which weapon item it should be using to attack, as it needs to pass the weapon item as a parameter when it creates a new AttackAction class, which extended from the abstract Action class.

REQ2: The Abandoned Village's Surroundings



The diagram represents the functionalities stated in requirement 2 of the assignment. In requirement 2, we have to deal with a ground where any entities will be killed if they had stepped into a bottomless pit called Void. Furthermore, there will be enemy spawned and attacks the player in the graveyard.

We introduced 3 new concrete classes and 1 new abstract class which are Void, Graveyard, WanderingUndead and Enemy abstract class. We added more classes and new packages. In the "actors" package, we add a new subpackage named "enemies", which stores all type of enemy in the game.

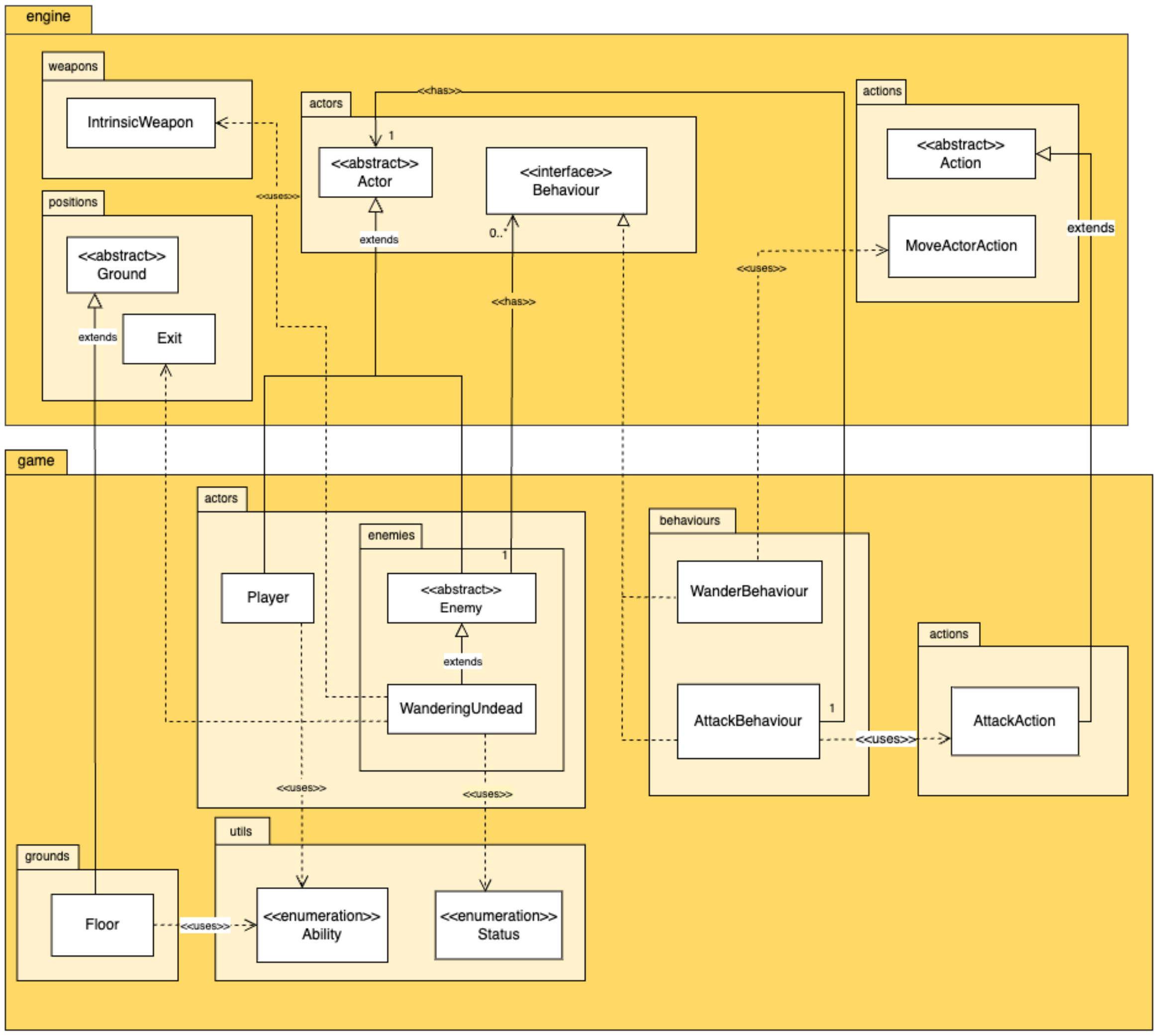
We adhere to the Liskov Substitution Principle (LSP) by abstracting enemy into a class, where all enemy entities can extend from that and prevents the need for checking their identities in each turn of game. It also helps in improving the extensibility of code, for adding more enemy into the system. However, it's worth noting that as the system grows larger and more complex, maintaining this code may become challenging.

We designed both the Void class and Graveyard class inherit from the abstract Ground class in our game engine to avoid repetition (DRY) as they share common attributes and methods. The relationship between Graveyard and abstract Enemy class with Player is association in which one of the Graveyard can have one Enemy spawned at each turn. Thus, the multiplicity of the association relation is exactly one. In the future implementation, this design decision reduces multiple dependencies (ReD) on different classes that is required, such as the WanderingUndead class.

The advantage of this design is that all methods in ground abstract class can be overridden to provide different implementations. This design adheres to the Open/Closed Principle (OCP) because the Void and Graveyard classes extend the functionality of the ground without altering existing code. Nonetheless, a possible drawback is that as the system expands in size and complexity, maintaining the code could become increasingly challenging.

The Void class has dependency on Display class, as Void class uses Display class to print a message from FancyMessage class to the output console to show the actor is killed when they stepped into bottomless pit. The application class also have dependencies on Player, Void, Enemy and Graveyard class as they are being initialized in Application.

REQ3: The Wandering Undead



The diagram represents the functionalities stated in requirement 3 of the assignment. In requirement 3, we have to deal with whether which entities can enter the floor, the enemy being spawn in graveyard, which is able to attack the player and wander around.

We introduced 2 new concrete classes such as WanderingUndead (enemies package) and AttackBehaviour (behaviours package).

There is new enumeration constant created in Ability class, ENTER_FLOOR, which is used to check if the actor can enter the floor. Thus establish the dependency between Ability class and Floor class.

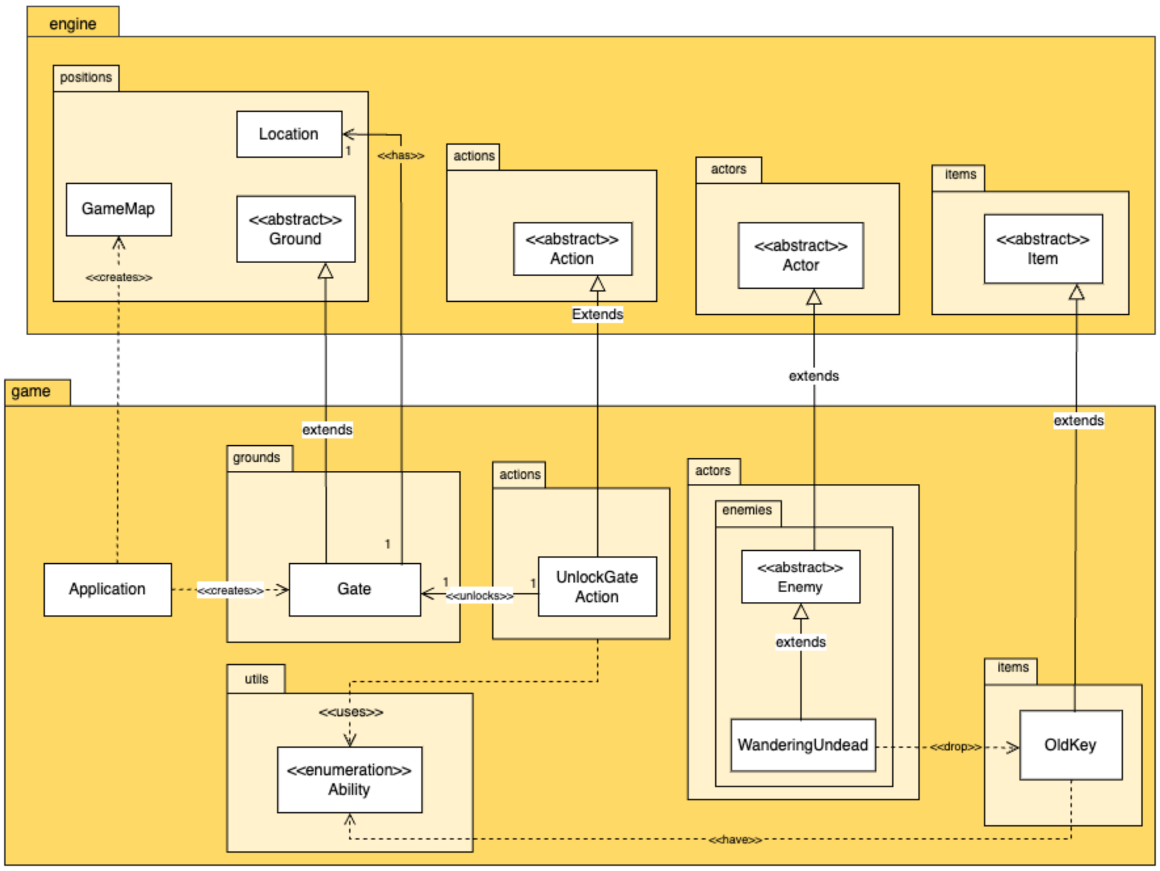
The WanderingUndead has dependency on IntristicWeapon class as the WanderingUndead can attack the player with its limbs, dealing 30 damage with 50% accuracy. Besides that, we designed a new Status enumeration constant, HOSTILE_TO_PLAYER, which is used to establish a dependency between the WanderingUndead class and the Status enumearation class. This relationship is intended to enable the WanderingUndead class to assess every actor it encounters prior to carrying out any actions upon them. This can avoid the enemy spawned attack another enemy. Also, the WanderingUndead class has dependency with Exit class to identify whether there are any other actors in its surronding. This information assists the enemy to determine how it should behave in response to that actor.

We created a new package called the "behaviours". Within this package, we introduced a concrete class called "AttackBehaviour" which implements the "Behaviour" interface in the engine. This class is responsible for handling actions related to attacks. By structuring it this way, we ensure that AttackBehaviour depends on the AttackAction, effectively separating the behavior from the action itself. This separation is crucial because behaviors are exclusive to NPCs (Non-Playable Characters). It enhances the design's clarity and allows us to easily modify or replace the attack behavior without affecting the AttackAction class. If we ever need to introduce another behavior that utilizes AttackAction in the future, the system can be extended seamlessly. The AttackBehaviour class encompasses the logic for determining an actor's action, hiding the complexities of location assessment and target selection. It offers a more accessible interface for obtaining the desired action. Additionally, it adheres to the Single Responsibility Principle (SRP) by encapsulating only the logic related to attack behavior, maintaining a clear focus on target evaluation and selecting appropriate attack actions.

Furthermore, the WanderBehaviour class will be responsible for controlling the movement of all enemies. Consequently, there exists a dependency relationship between the WanderBehaviour class and the MoveActorAction within the engine.

Additionally, abstract Enemy class is associated with Behaviour interface because we need an attribute in Enemy class to store all the behaviours that the enemy can have. One enemy can have zero or more behaviour, such as attacking the player, wander around. This can help to avoid code duplication (DRY).

REQ4: The Burial Ground



The diagram represents the functionalities stated in requirement 4 of the assignment. In requirement 4, we have to deal with a new region of the abandoned village, which is known as the burial ground. The player will not be able to proceed without a key since the gate is locked.

We introduced new concrete classes such as Gate (grounds package), UnlockGateAction (actions package) and OldKey (items package).

The Application class has a dependency relationship with the GameMap class and the Gate class, as it will create a new GameMap instance and Gate instance. The WanderingUndead class also has dependency relationship with the OldKey class, as it has certain rate to drop the OldKey instance once it is defeated by the player.

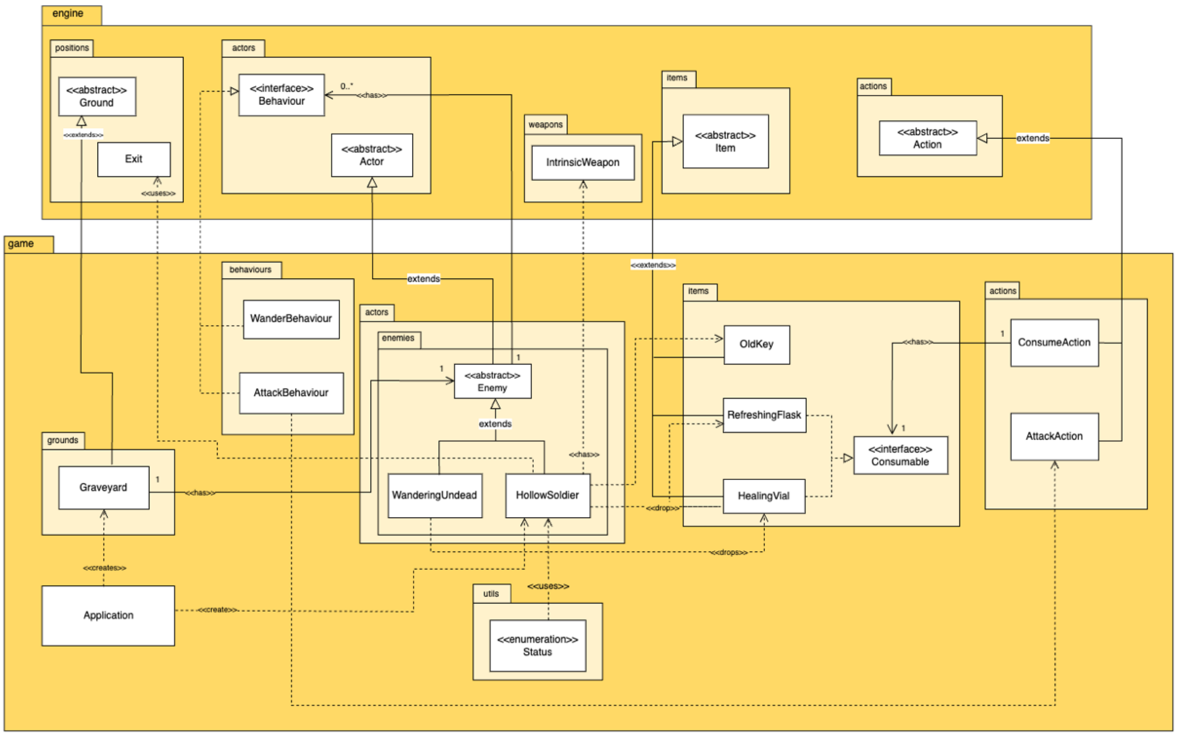
We chosed to make the Gate class extend from the abstract Ground class so that it can inherit and utilize the methods from its base class (Ground). This decision enables the Gate class to access the allowableActions method in the Ground class. The allowableActions method serves two purposes: it returns an action to unlock the gate when the actor's item inventory contains the OldKey item, and it provides a MoveActorAction method for the actor to teleport to a different location. Consequently, Thus, it results in a dependency relationship with UnlockGateAction class.

Moreover, Gate class forms an association relationship with the Location class from the engine, facilitating the teleportation of the Actor to the desired destination. This design makes our system more adaptable and easier to maintain when creating additional instances of Gate classes for assisting Actors in traveling to different locations.

The UnlockGateAction class inherits from the abstract Action class from the game engine. It has association with Gate as it needs to know which Gate instance to be open, as it needs to pass the Gate instance as a parameter when it creates a new UnlockGateAction. Moreover, we also introduced a new enumeration constant, OPEN_GATE in Ability enumeration class. This ability is added to the OldKey class. When we call the ability of OPEN_GATE, it will check if the item in actor inventory has capability of this status. If it is, it will call the unlock method in Gate class, which set the gate status to false as indicating gate is unlocked. Hence, Ability enumeration class will have the dependency relationship with OldKey and UnlockGateAction. This design allow new features can be seamlessly integrated into the UnlockGateAction class, ensuring it adheres to the Single Responsibility Principle (SRP) by assigning each class a single reason for modification. This approach also fosters the creation of reusable code for future game implementations, making our system easily extensible. One drawback of this design choice could be the potential complexity in the code for item verification.

The OldKey class is also interconnected with the Ability enumeartion class, forming a dependency. In future implementations, this connection can be utilized to determine if an Actor possesses an instance of the OldKey class. This approach serves to minimize the dependency on the OldKey class and prevents potential violations of the Open/Closed Principle (OCP) in the event of new Item classes being introduced that may also require a dependency with the Gate class. Consequently, we do not have to rewrite existing code within the Gate class to verify the presence of an OldKey instance in the Player's inventory. This design adheres to the Open/Closed Principle (OCP).

REQ5: The Inhabitants of The Burial Ground



The diagram represents the functionalities stated in requirement 5 of the assignment. In requirement 5, we have to deal with a different type of enemy in the graveyard of burial ground. There will be certain rate for an enemy to drop HealingVial and RefreshingFlask items once it is defeated at every turn each turn of the game.

We introduced 3 new concrete classes such as HollowSoldier (enemies packages), HealingVial (items package) and RefreshingFlask (items package). HealingVial, RefreshingFlask shares common attributes and methods with Item, it is logical to extend it from Item abstract class to avoid repetition (DRY). HollowSoldier also share common attributes and methods with Enemy, Action, similar to WanderingUndead, hence we also extend them from Enemy abstract class.

The Application class establishes a dependency with the Graveyard class on various game maps, as well as the HollowSoldier class, as it creates an instance of it in game map. This is shown in previous UML diagram in terms of WanderingUndead. As mentioned before, This HollowSolider class will have a dependency relationship with the Status enumeration class to assess every actor it encounters prior to carrying out any actions upon them. Also, the HollowSolider class has dependency with Exit class to identify whether there are any other actors in its surronding. This information assists the enemy to determine how it should behave in response to that actor. Additionally, abstract Enemy class is associated with Behaviour interface because we need an attribute in Enemy class to store all the behaviours that the enemy can have. One enemy can have zero or more behaviour, such as attacking the player, wander around. This can help to avoid code duplication (DRY).

The WanderingUndead class has dependency relationship with the HealingVial class, similarly the HollowSolider class also has dependency relationship with the OldKey class, HealingVial class and RefreshingFlask as it has certain rate to drop the item instance once one of them is defeated by the player.

We created an interface called Consumable. It is used to stores methods that represent the consume effect of the item for the actor. It is implemented by HealingVials class and RefreshingFlask class as both class possess healing function to the item's holder if consumed. In future implementations, if there is need to add any new kind of consume effect of the item, the system can be extended seamlessly. By doing so, we aligned with the DRY principle.

In addition, we also introduced a ConsumeAction class that extended from Action abstract class from the engine, which will take care of all the actions when an actor consumes something consumable. The ConsumeAction has an association relationship with the Consumable interface class, meaning that any class implementing Consumable can be pass to the ConsumeAction constructor. This allows for differentiation among these classes and the execution of distinct methods before returning results to the engine. Ultimately, this helps avoid issues like code smells, such as the need for downcasting, by enabling the use of polymorphism.