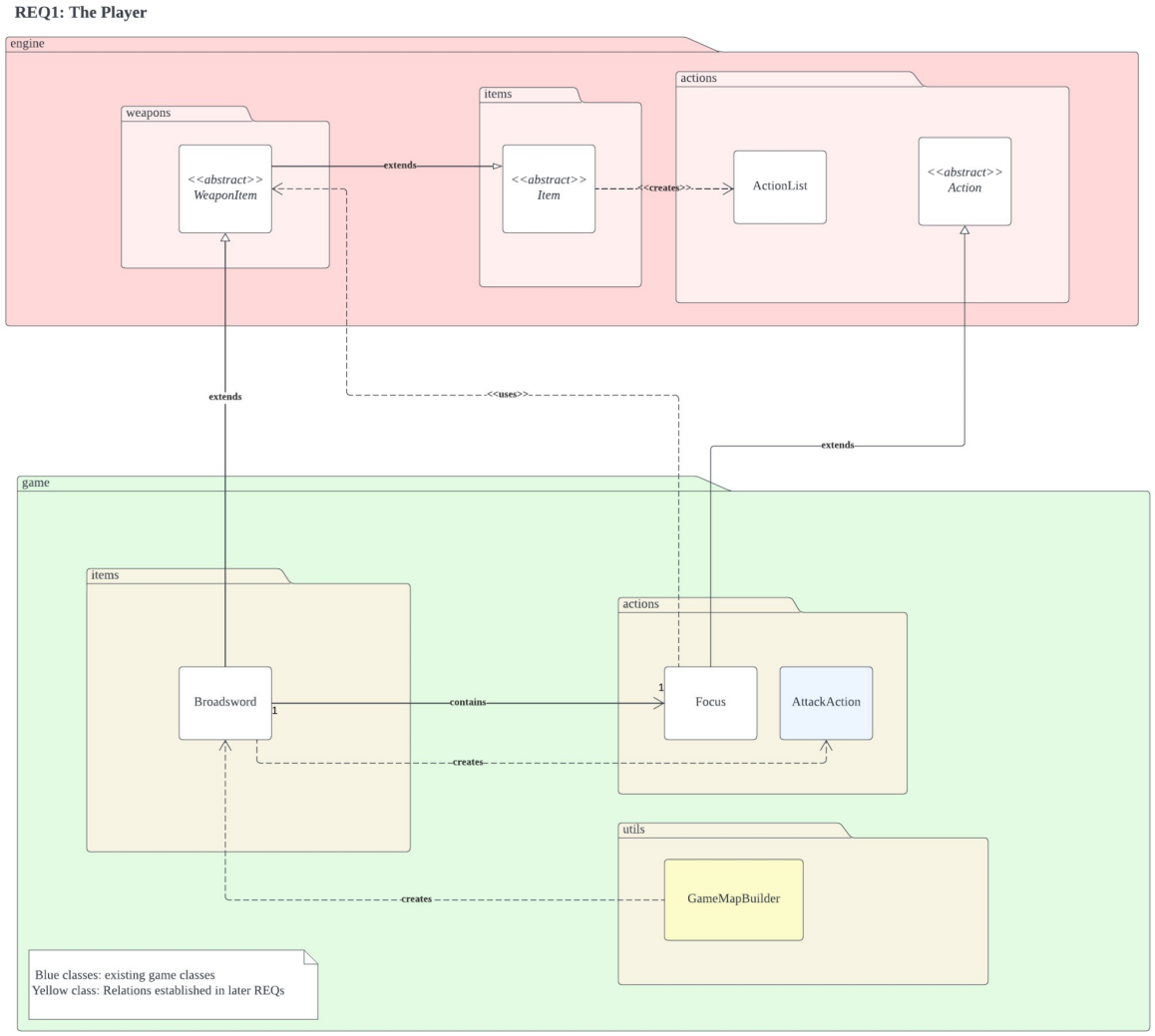
Sandra Lo Yii Shin (33338604)

FIT2099 Assignment 1
Design Rationale

REQ1: The Player



**GameMapBuilder (yellow box) and its relations to other classes implemented in REQ4

Tasks

- Display player stats before printing the console menu
- Update player's intrinsic weapon stats, allow player to recover stamina by 1% of max stamina per turn.

- Create a starting weapon (*Broadsword)* that can be picked up by the player, with specific stats and a special skill *('Focus')*

Player

Player has a *displayStatus* method called at the beginning of each *playTurn* method iteration, returning a string displaying their name, HP and Stamina.

Stamina recovery is done through a dedicated *recoverStamina* method.
The *getIntrinsicWeapon* from the *IntrinsicWeapon* class is used to set new stats for the player's basic attack.

Considering these are player-related functionalities and that there is only one playable character, this implementation adheres to the requirements while keeping the code simple. If there were to be different stamina/health recovery mechanics for different playable characters in the future however, the player class may have to delegate the customization to other classes to ensure extensibility and maintainability, as stated by SRP.

Broadsword

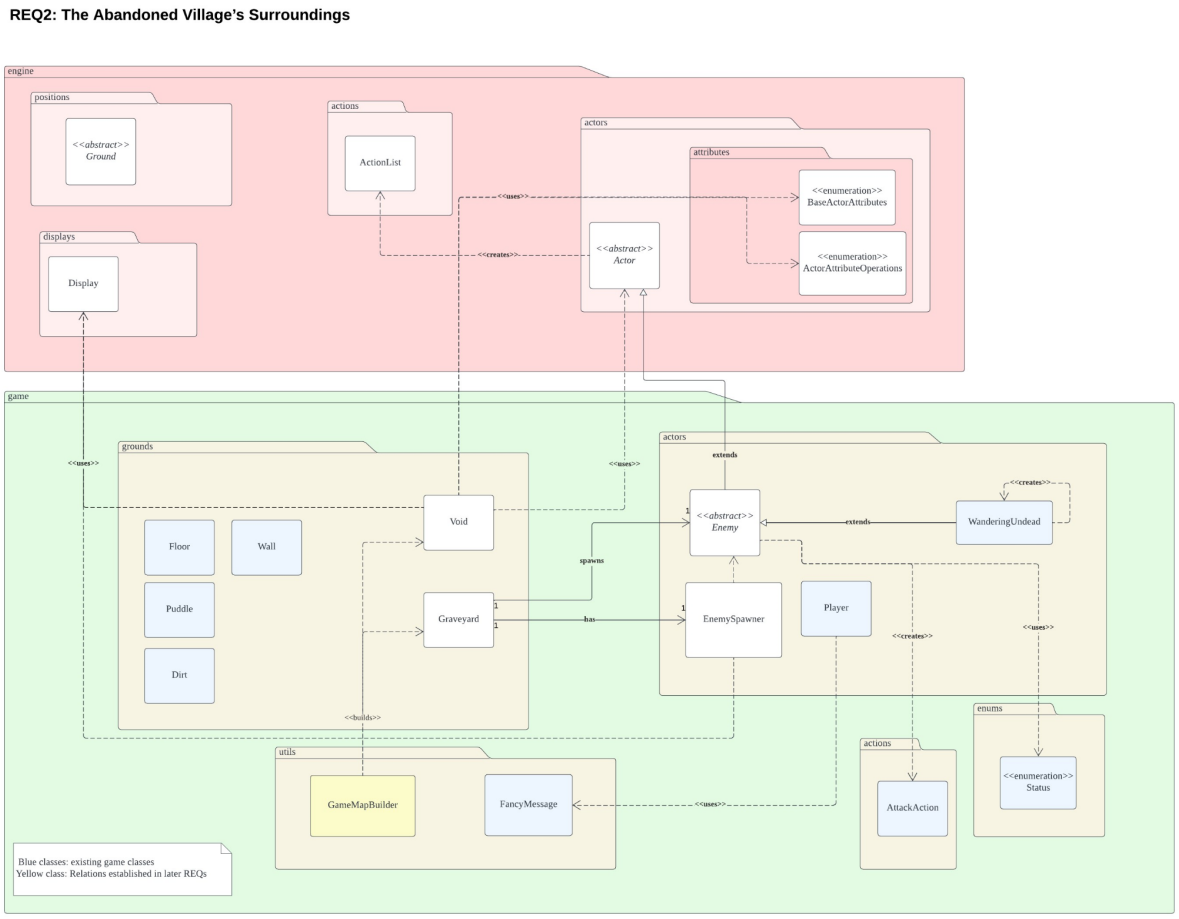
The *Broadsword* extends the abstract *WeaponItem* class, thus containing base weapon features and subsequently, item features (allowing it to be picked up and dropped) and allows for addition and modification of the weapon without code duplication. (DRY, OCP, LSP)

This weapon uses an *AttackAction* to allow the user to attack with it.

Focus is implemented as a concrete class extending engine code's *Action* abstract class, since it should allow the user to select when to activate this skill, consistent with the provided implementations of other actions in the game. *Broadsword* thus also contains a single *Focus* instance to keep track of turn, using its *tick* method.

The skill takes effect for 5 turns upon activation, but it is assumed that there may be more powerful weapons that provide different values for its effects. *Broadsword* takes responsibility for managing the skill's activity, while *Focus* deals with its effects, thus centralising the control logic within the weapon class. (SRP)

REQ2: The Abandoned Village's Surroundings



**GameMapBuilder (yellow box) and its relations to other classes implemented in REQ4

Tasks

- Implement a new ground type, *Void*, that kills any entity that steps into it
- If the player dies, print the fancy message provided
- Implement a new ground type, *Graveyard*, that spawns a WanderingUndead with 25% chance per turn

Void

Void is implemented as a subclass of *Ground*, consistent with the other ground types provided in the base code, following OCP that allows new Ground types to be made without modifying the base class.

Using its tick method, it keeps track of turns and eliminates entities entering the *Void*. This presents as an immediate consequence of the actor's actions before they get to play their

next turn, like an actual game. Here, the actor's *unconscious* method is called to eliminate them.

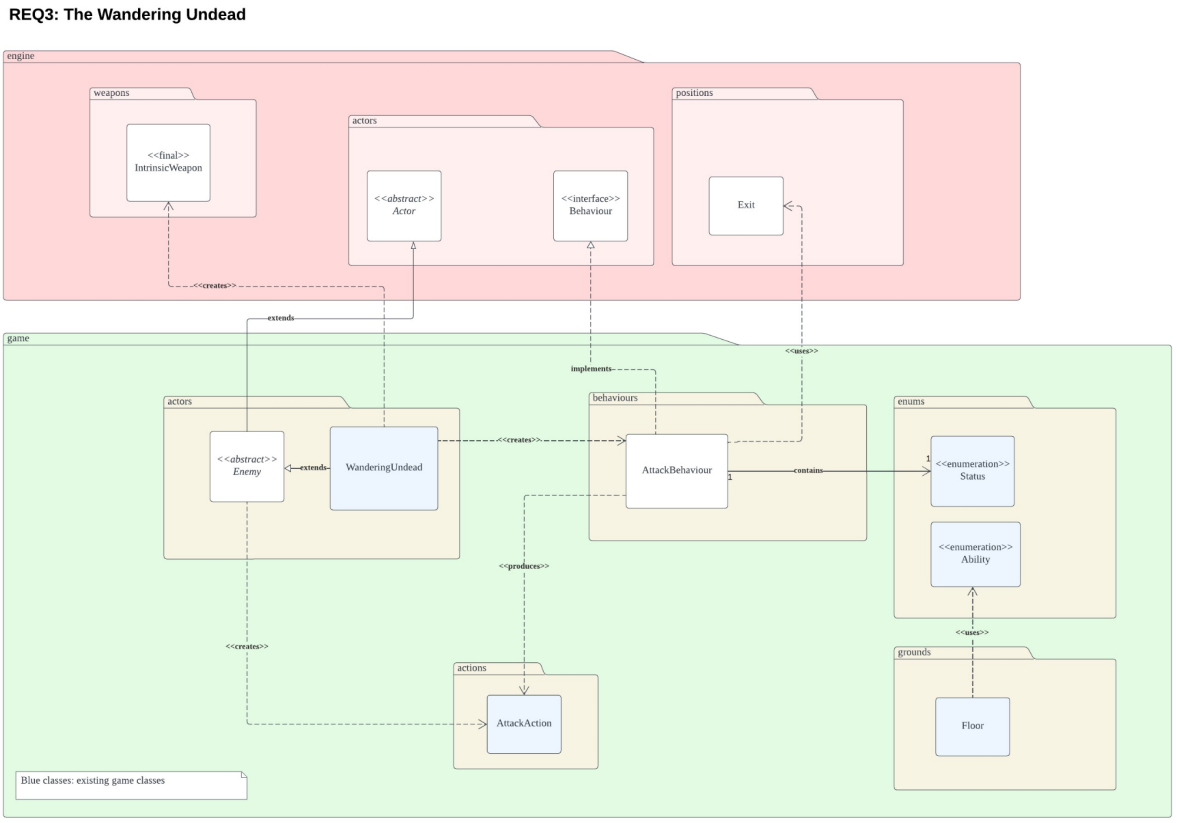
For the player, *unconscious* methods are both overridden to implement the fancy message display through a *playerDeadMessage* method in the *Player* class, before eliminating the player. This maintains the expected behaviour of the actor upon death while adding a player-specific functionality (LSP).

Graveyard, Enemy, EnemySpawner

Graveyard includes a new *EnemySpawner* instance responsible for handling the spawning process. This thus separates the spawning logic from the ground type itself (SRP).
This accounts for the consideration that there can be more than one spawning grounds in the game, each having different spawning mechanics, eg. when more than one actor is spawned at once, or that the same ground type can have different mechanics in itself. To achieve this, the spawn rate is specified as a parameter passed into *Graveyard*'s constructor.

WanderingUndead, and all enemies, extend an *Enemy* abstract class, a subclass of *Actor.*
Though this adds to the class hierarchy, potentially increasing code complexity, it collects all common functionality in an enemy, and is not expected to be further extended for this game, such that the identity between player and enemy is distinguishable. Each enemy has a method to return an instance of itself to be used by a spawner or any other enemy-specific functionalities.

REQ3: The Wandering Undead



Tasks

- Implement the attack feature of *WanderingUndead,* who can only attack the player but not another enemy
- Update the intrinsic weapon stats of the *WanderingUndead*
- *WanderingUndead* cannot enter a floor

Actor

WanderingUndead

Similar to the player in REQ1, *getIntrinsicWeapon* is used to update the basic attack of the *WanderingUndead*.

AttackBehaviour, Enemy

From the given code, Behaviours intend to help NPCs perform actions. Thus this feature is realised through a *AttackBehaviour* class, aligning with the intentions of encapsulating the logic related to the tendency to attack by the NPC (SRP). An *AttackAction* can be returned depending on the targetCapability it was set to check for. This allows for flexibility in which

the Enemy can attack specific targets only. If needed, enemies can also attack differently other than a basic attack, such as using a weapon, or attack through spawns. (OCP)

It is now assumed that all enemies can be attacked by the *Player* who is currently *HOSTILE_TO_ENEMY* (targetCapability).

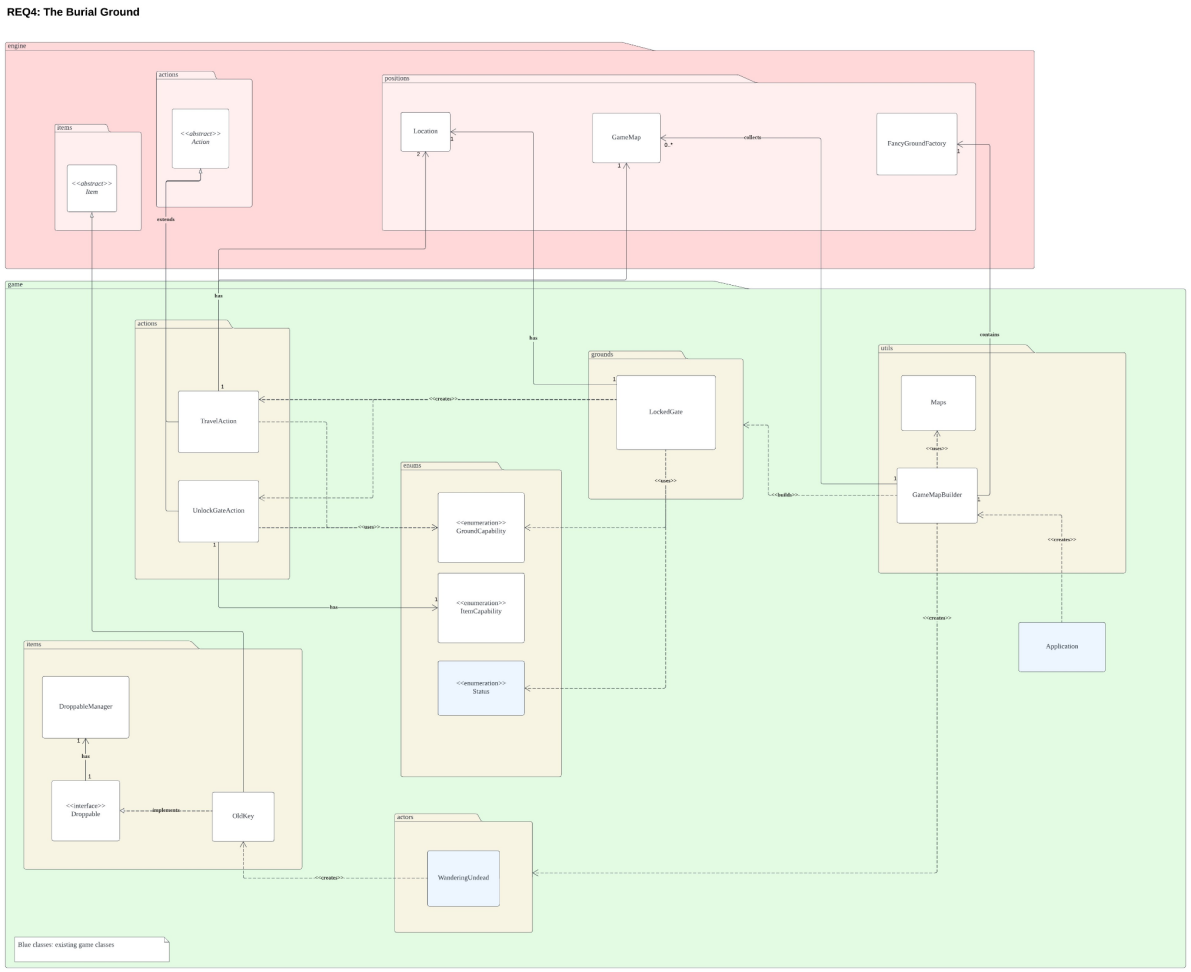
Enums

Ability

A *WALK_ON_FLOORS* ability is added to the actor's capability list. Currently, only the player has this ability, assuming that there might be an enemy or a NPC that can walk on floors in the future, like humans or pets.

The *Floor* class' *canActorEnter* method is overridden to check for this ability.

REQ4: The Burial Ground



Tasks:

- Generate a new map for the burial ground
- Implement a locked gate with an Old Key required, dropped by the WanderingUndead
- No one can enter the gate when locked, and when unlocked anyone can step in, but only the player can travel with it.
- Add another gate in the Burial Ground to go back to the Abandoned Village Map

<u>GameMapBuilder, Maps</u>

As the requirement calls for a new game map, due to increased complexity of the game, the map feature is now managed through two new respective classes, GameMapBuilder and Maps, instead of having the application handle the map generation. (SRP)

This thus diverts the dependencies of all objects that can be generated in the game from Application to GameMapBuilder. The game map strings are stored in Maps for quicker access to the location coordinates, and for ease of visualisation of the game flow.

This also promotes extensibility in which any new maps can be quickly created and accessed without major changes to the code. (OCP)

The creation of player and world however remains in the *Application* class as the game map builder mainly handles map related functionalities. (SRP)

LockedGate, UnlockGateAction, TravelAction, GroundCapability

The *LockedGate* is implemented as a Ground type as it can restrict actors from stepping on them before a requirement is fulfilled. It has a capability of *BLOCKED* to indicate its temporarily blocked state, which is then removed if the gate is unlocked by *UnlockGateAction* passed by an item that can unlock the gate. The use of an action allows the user to interact with the gate before unlocking it, to make the player be aware of the presence of a gate.

Specific to the gate, once unlocked, it gains the ability to be *TRANSPORTABLE,* thus allowing *TRAVEL*-capable actors to transport to a new map, ie. *LockedGate* can return a *TravelAction* to the *TRAVEL*-capable actor.

With the use of Capabilities, this opens up options for the actor to ‘break locks’ using different items other than a key. This can also allow timed gates to be implemented, eg. The gate locks itself up after a number of turns.

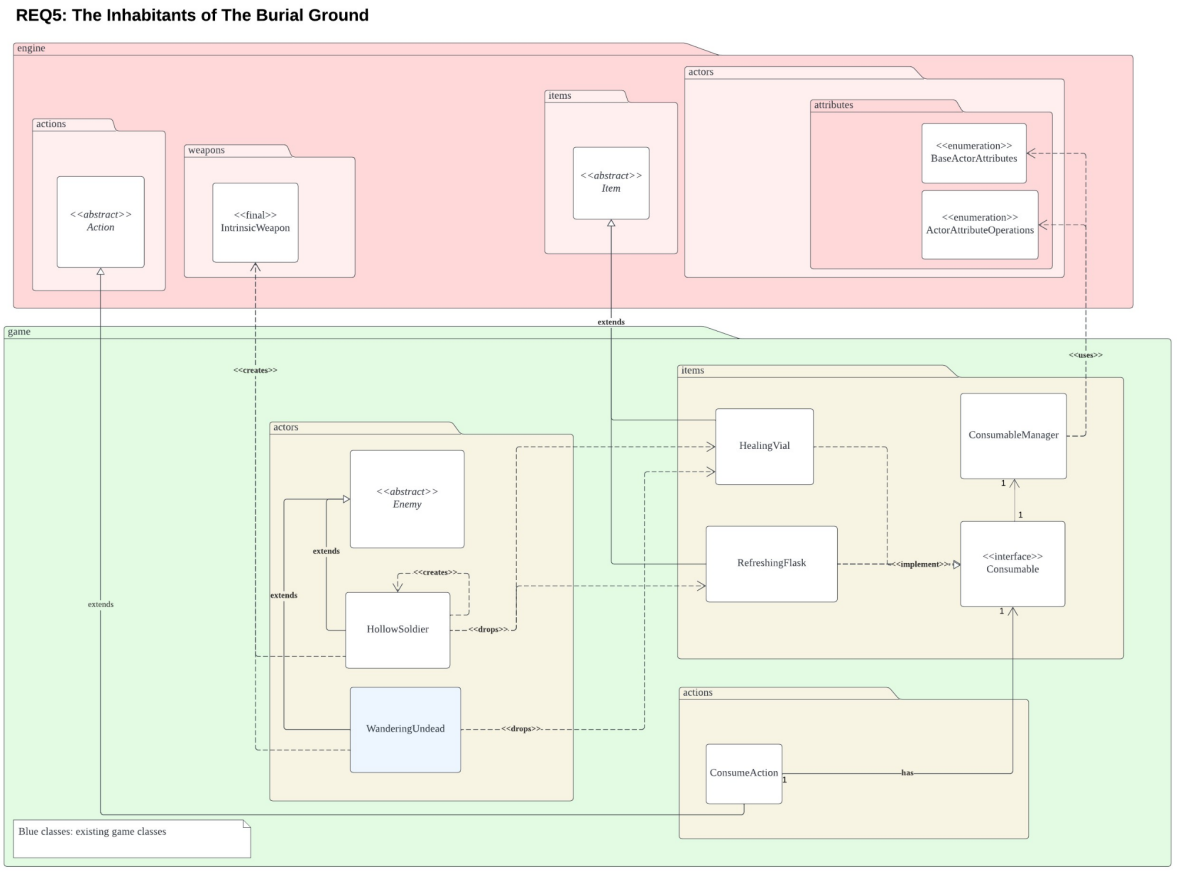
OldKey, ItemCapability, Droppable, DroppableManager

OldKey is a droppable item that can *UNLOCK_A_GATE*, hence an interface *Droppable* and its manager is created to collect and manage all common functionalities of a droppable item. The manager collects common functionality of an item dropped action at a specified drop rate, and by a certain actor. This is to reduce code duplication since it is expected that items will have similar drop mechanics. (DRY)

Items that can be dropped by the actor will be implemented in their respective *unconscious* method, since they will only drop the item when slain by the player. The use of ItemCapabilities provides a clear way to define the behaviour of the item.

Though this creates more abstractions in the code, this ensures the reusability of a drop mechanism as compared to inline implementation in the item itself. The use of an interface also enables the item to demonstrate different functionalities and separates it from the inheritance hierarchy.

REQ5: The Inhabitants of The Burial Ground



Tasks

- Implement a new enemy, Hollow Soldier that can be spawned from the graveyard
- Hollow soldier can drop a healing vial at a 20% chance, and a refreshing flask at 30% chance
- The drops are consumable, and not reusable after consumption
- Allow the WanderingUndead to drop a healing vial at 20% chance as well
    - Drop probabilities can be independent of each other

HollowSoldier

With the use of the Enemy abstract class, a HollowSoldier is extended, similar to WanderingUndead, and like WanderingUndead, it has a WanderingBehaviour and an AttackBehaviour. However, these are not both abstracted to become a common functionality in the Enemy class, since some enemies will not develop a tendency to attack before being provoked, or may stay idle guarding something precious.

This enemy can be spawned via *Graveyard* in the new map. Due to the implementation in REQ2, this can be done directly by passing an instance of itself to the spawner, and its spawn rate to the *Graveyard* constructor, demonstrating its extensibility.

Consumable, ConsumableManager, RefreshingFlask, HealingVial, ConsumeAction

Consumable and its manager class contribute to the implementation of the consume item function. Like *Droppable*, this aims to demonstrate the functionality of a consumable item. The manager class, again, collects said functionality, and removes the item after consumption. Item consumption can be selected by the user through a *ConsumeAction* provided by the consumables. This ensures that the item can only be consumed after being picked up.

Despite the similar mechanics of consuming a stamina item and a healing item, these have been implemented in separate methods in the manager class to reduce further abstractions for the time being in favour of simplicity and code clarity. If the consumable, say modifies two attributes of the actor at once, and provides a side effect as a result, then it would be more ideal to have the target attribute checked and perform the consumption with the target attribute passed as a parameter when called. The manager is still extensible in which different mechanics of attribute modification (eg. health increased by fixed points, items can be used more than once) can be implemented with no changes to the other code.

RefreshingFlask, HealingVial implement these classes and like *OldKey*, is created in the overridden *unconscious* methods of their respective dropping enemies, since both of these items can also be dropped (implementing *Droppable*) in addition to being consumable (ISP).