Design Rationale

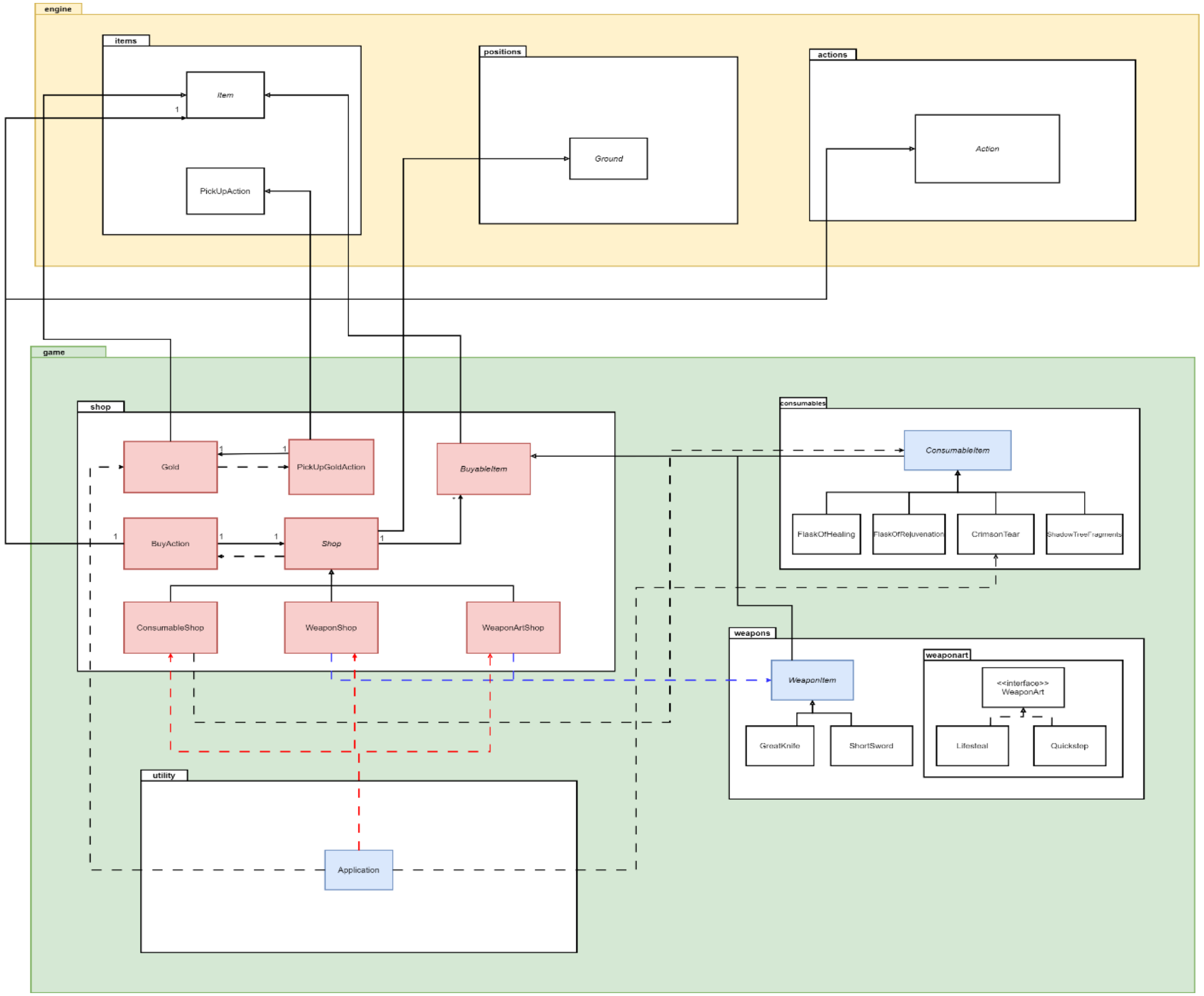
REQ4: CREATIVE MODE

General Ideas implemented:

1) Gold and Shop system
    - A new item known as Gold acts as a new currency and can be picked up by Player to add balance to their wallet
    - Gold is randomly spawned on the Gravesite Map and has different amount
    - Different enemies also add different balance to Player's wallet when they are slayed
    - Player can spend their wallet balance at a new ground type called Shop.
    - There are 3 different types of shops namely, ConsumableShop, WeaponShop and WeaponArtShop.
    - Player can buy ConsumableItems from ConsumableShop, WeaponItems from WeaponShop and WeaponItems with WeaponArt from WeaponArtShop on the Gravesite Map.
    - Items that are bought at the shop will be added into Player's inventory and can be used.
    - All the Shops are located at the top right of the Gravesite Map.

2) Pet System
    - A new actor know as Pet is an actor that follows the Player around and have special behaviours depending on the type of pet.
    - However, Pets can only follow Player around in Gravesite Map and cannot travel to the other maps.
    - Player can only choose 1 pet to be its companion throughout the game.
    - There are 3 different types of pets namely, LightningEel, HealingUnicorn and TankyKong.
    - LightningEel behaves such that it can attack all enemies within its surroundings dealing heavy damage, but LightningEel has very little hit points.
    - HealingUnicorn behaves such that it can heal the Player every turn as long as Player is within its surroundings.
    - TankyKong behaves such that it can deal a light attack (low damage attack) to 1 enemy within its surrounding, but TankyKong has very high hit points.
    - All the Pets are located at the top left of the Gravesite Map.

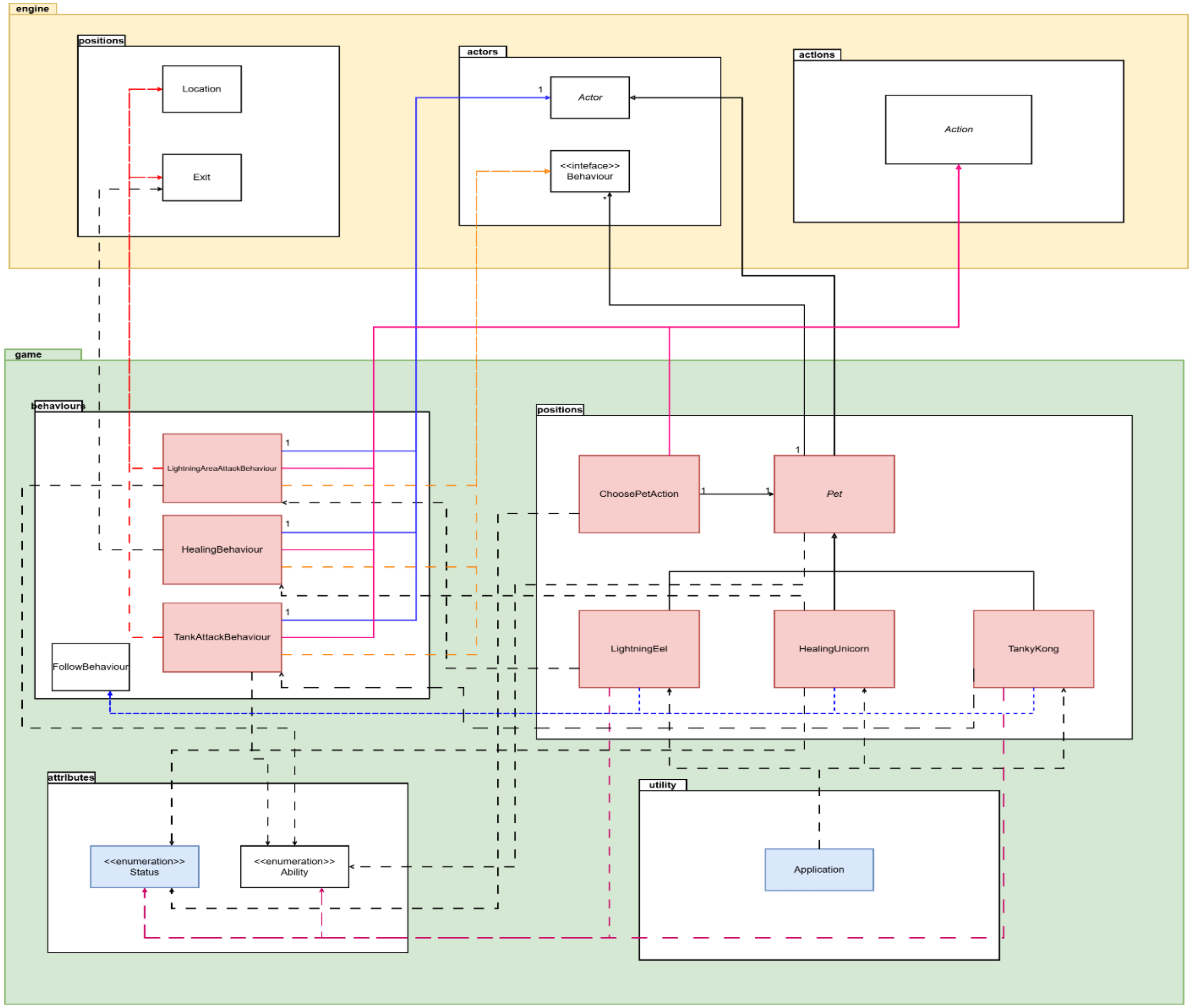
UML Diagram for Gold and Shop system implementation



Modified Classes = Light Blue

Classes Created = Light Pink

UML diagram for Pet system implementation



Modified Classes = Light Blue

Classes Created = Light Pink

Rationale for Idea 1 (Gold and Shop system)

Classes Created / Modified	Roles and Responsibilities	Rationale
Created a Gold class (extends Item)	Class representing Gold item. Relationship: 1. Extends Item to achieve Item functionality. 2. Has dependency with PickUpGoldAction to allow gold to be picked up. 3. Depended on by Application to randomly spawn on gold on Gravesite Map.	Reasons for decision: 1. Single Responsibility Principle (SRP): This class focuses solely on representing a gold item that adds balance to the Player's wallet when picked up. This makes the code easier to understand and debug. 2. It promotes easy expansion as in the future if gold has additional benefits to be given to the Player, we can easily expand the functionality of Gold without modifying any other existing classes.
Created a PickUpGoldAction class (extends PickUpAction)	Class representing picking up gold action. Relationship: 1. Extends PickUpAction to achieve PickUpAction functionality 2. Has association with Gold to pick up gold	Reasons for decision: 1. Liskov Substitution Principle (LSP): Since PickUpGoldAction extends PickUpAction, it can replace any instance of PickUpAction without modifying the program's intended behaviour as it only adds specific functionality related to gold but maintains expected behaviour of picking up an item. 2. Open-closed Principle (OCP): PickUpGoldAction extends PickUpAction which is a general action for picking up items. This allows classes to modify the pickup behaviour such as PickUpGoldAction will add balance to Player's wallet without changing the core functionality of the PickUpAction class. In the future, if there are more PickUpActions with different pick up behaviour, we can simply extend and add the new behaviour without modifying any existing classes.

Created a BuyableItem abstract class (extends Item)	Class representing buyable items. Relationship: 1. Extends Item to achieve Item functionalities.	Reasons for decision: 1. Creation of BuyableItem allows us to define the blueprint of an item that can be bought. The abstract method getPrice() ensures that all subclasses that extends BuyableItem will have to define the implementation of getPrice() to get the price of the item. 2. Future Extensibility – ConsumableItem and WeaponItem class extends BuyableItem as they can be bought at the shop. In the future, if there are more types of Items that can be bought from the shop such as ArmorItem, I can simply extend the BuyableItem class without re-implementing the purchasing logic. 3. Avoids design smells such as Large Class Smell – We avoided the creation of a “God” class that handles all responsibilities. Instead, the BuyableItem class focuses on the buyable aspect whereas ConsumableItem and WeaponItem focus solely on their specific functionalities such as consuming items or using weapons.
Modified ConsumableItem class to extend BuyableItem	Class representing consumable items. Relationship: 1. Extends BuyableItem to achieve Item and buyable functionalities. 2. Implements Consumable to achieve consumable functionalities.	
Modified WeaponItem class to extend BuyableItem	Class representing weapon items. Relationship: 1. Extends BuyableItem to achieve Weapon and buyable functionalities. 2. Implements Weapon to achieve weapon functionalities.	
Created a BuyAction class (extends Action)	Class representing buy action. Relationship:	Alternate Design Pattern: Creating a single AllItemShop class that handles all buying logic of different item types such as ConsumableItem and WeaponItem.

	1. Extends Action abstract class to achieve action functionalitles. 2. Has association with Item to conduct the BuyAction on the item. 3. Has association with Shop to allow Player to buy items from the shop.	Disadvantages of alternate design pattern: 1. Violates Single Responsibility Principle (SRP): The combined class now has to handle multiple responsibilities which is to manage the adding of ConsumableItem to the shop as well as WeaponItem. This also increases the likelihood of bugs and makes debugging more difficult because the class handles multiple functionalities. 2. Violates Open-Closed Principle (OCP) – When new types of Shops are to be introduced, instead of extending from the Shop abstract class, we now need to make changes in the AllItemShop. This results in the AllItemShop class being not open for extension and not closed for modification. Finalised Design Pattern: Creating a ConsumableShop, WeaponShop and WeaponArtShop respectively that extends the Shop abstract class to allow the shops to have their own implementation but still has the core functionality of a Shop. Reasons for Decision: 1. Single Responsibility Principle (SRP): The ConsumableShop focuses solely on selling ConsumableItem while the WeaponShop focuses solely on selling WeaponItems. The WeaponArtShop also focuses solely on selling WeaponItems with WeaponArts. By adhering to this principle, If there were changes to be made to ConsumableShop, I can easily modify
Created a Shop abstract class (extends Ground)	Class representing shop Relationship: 1. Extends Ground to achieve Ground functionalities. 2. Has association with BuyableItem to sell buyable items 3. Has dependency with BuyAction to allow Player to buy items from Shop.	
Created a ConsumableShop (extends Shop)	Class representing consumable shop Relationship: 1. Extends Shop to achieve Shop functionalities. 2. Has dependency with ConsumableItem to add	

	consumable items to the shop	the ConsumableShop class without affecting the WeaponShop and WeaponArtShop. 2. Open-Closed Principle (OCP): The extension from the Shop abstract class allows for new shops to be created in the future such as ArmorShop by simply extending from the Shop abstract class without modifying any existing classes. 3. Liskov Substitution Principle (LSP): BuyAction class extends from Action abstract class. By doing this, we can substitute Action with BuyAction without modifying the intended behaviour of the program. 4. Dependency Inversion Principle (DIP): All the different Shops extends the Shop abstract class. The Shop abstract class is a high-level module as it is an abstraction. By using this, we can reduce the tight coupling as subclasses of Shops have their respective implementation. 5. Interface Segregation Principle (ISP): The subclasses of Shop only extends Shop abstract class and only implements methods relevant to Shop. It does not rely on any other unnecessary methods or interfaces and only handles shop-related logic. 6. Don’t Repeat Yourself Principle (DRY): Since the 3 different shops extend the Shop abstract class, we can avoid duplication of common functionalities such as the allowableActions() method as it has already been implemented in the Shop abstract class.
Created a WeaponShop (extends Shop)	Class representing weapon shop Relationship: 1. Extends Shop to achieve Shop functionalities. 2. Has dependency with WeaponItem to add weapon items to the shop	
Created a WeaponArtShop (extends Shop)	Class representing weapon art shop Relationship: 1. Extends Shop to achieve Shop functionalities. 2. Has dependency with WeaponItem to add weapon items with weapon art to the shop	
Modified Application class to create new instance of ConsumableShop, WeaponShop and WeaponItemShop to be placed on the Gravesite Map.	Class representing the Application Relationship: 1. Has dependency with ConsumableShop, WeaponShop and WeaponItemShop to create new instance.	

		7. Manages Connascence of Type by depending on abstractions: The itemsForSale list in the Shop class can only contain BuyableItem objects. Only BuyableItem objects can be added to the Shop. This creates connascence between the Shop class and its child classes such as ConsumableShop which must also only add BuyableItem objects to their shop. 8. Avoids design smells such as Shotgun Surgery: This design smell occurs when a small change requires modifications across multiples classes. By inheriting from Shop, if changes are to be made to Shop logic such as allowableActions logic needs to be changed, I can simply modify the allowableActions in the Shop abstract class and the changes will automatically apply to all child classes. Disadvantages of Finalised Design Pattern: 1. The creation of additional classes can lead to memory overhead and increase the complexity of the program, especially as the number of classes grows significantly when there are new types of Shops. 2. The code that is very modular increases complexity and when a bug arises, the debugger needs to have good understanding on the interactions between abstractions and classes in order to debug.

Rationale for Idea 2 (Pet system)

Classes Created / Modified	Roles and Responsibilities	Rationale
Created a LightningAreaAttackBehaviour class (extends Action implements Behaviour)	Class representing a LightningAreaAttack behaviour. Relationship: 1. Extends Action to achieve Action functionalities 2. Implements Behaviour to achieve behaviour functionalities such as getAction(). 3. Has association with actor to perform the attack on the actor. 4. Has dependency with Location and Exit to determine the surroundings. 5. Has dependency with Ability to check if actor has the Ability.IMMUNE capability meaning friendly unit.	Reasons for decision: 1. Single Responsibility Principle (SRP): Each class namely LightningAreaAttackBehaviour, HealingBehaviour and TankAttackBehaviour focuses on its own respective behaviour whether it is to execute an area attack, heal the player or attack a single adjacent enemy. This makes the code more maintainable as well as makes it easier to understand for new developers. 2. Open – closed Principle (OCP): By separating the behaviours into different classes, if I want to change how the healing in HealingBehaviour works, I can modify HealingBehaviour only without affecting other behaviours. This means that it supports extension of the game logic without modifying existing functionalities. 3. Interface Segregation Principle (ISP): Each class only implements the interface that is related which in this case is the Behaviour interface. It only overrides relevant methods from the Behaviour interface and not use any other methods that are not related to its functionalities, minimizing dependencies. 4. Don't Repeat Yourself Principle (DRY): In each class the use of getExits() avoids the needs to rewrite the logic for finding adjacent locations reducing code duplication.
Created a TankAttackBehaviour class (extends Action implements Behaviour)	Class representing a TankAttack behaviour. Relationship: 1. Extends Action to achieve Action functionalities	

	2. Implements Behaviour to achieve behaviour functionalities such as getAction(). 3. Has association with actor to perform the attack on the actor. 4. Has dependency with Location and Exit to determine the surroundings. 5. Has dependency with Ability to check if actor has the Ability.IMMUNE capability meaning friendly unit.	5. High Connascence of Execution as order is crucial for proper logic: In LightningAreaAttackBehaviour and TankAttackBehaviour, enemy.hurt() needs to be called first before enemy.isConscious(). This specific sequence is important to ensure that enemy will need to take damage before checking its consciousness. 6. Avoids design smells such as Feature Envy: In each behaviour class, the logic that applies to the target actor such as checking for capability Ability.IMMUNE or updating actor's health is left to the actor and the behaviour classes themselves do not directly manipulate the target actor's internal functionalities.
Created a HealingBehaviour class (extends Action implements Behaviour)	Class representing a Healing behaviour. Relationship: 1. Extends Action to achieve Action functionalities 2. Implements Behaviour to achieve behaviour functionalities such as getAction(). 3. Has association with actor to perform the healing on the actor. 4. Has dependency with exit to determine its surroundings.	

Created a ChoosePetAction class (extends Action)	Class representing choose pet action. Relationship: 1. Extends Action to achieve action functionalities. 2. Has association with Pet to conduct action on Pet. 3. Has dependency with Status to check if actor has Status.HAS_PET and update Status.CHOSEN_PET.	Alternate Design Pattern: Creating a single AllinOnePet class that handles all Pet logic of different Pet types such as LightningEel, TankyKong and HealingUnicorn. Disadvantages of Alternate Design Pattern: 1. Violates Single Responsibility Principle (SRP): The combined class now has to handle multiple responsibilities which is to manage the adding of different behaviours depending on the type of Pet. 2. Violates Open-Closed Principle (OCP) – When new types of Pets are to be introduced, instead of extending from the Pet abstract class, we now need to make changes in the AllinOnePet class. This results in the AllinOnePet class being not open for extension and not closed for modification. Finalised Design Pattern: Creating a LightningEel, TankyKong and HealingUnicorn respectively that extends the Pet abstract class to allow the pets to have their own implementation but still has the core functionality of a Pet. Reasons for decision: 1. Single Responsibility Principle (SRP): LightningEel, TankyKong and HealingUnicorn classes focus solely on defining their specific attributes and adding their respective
Created a Pet abstract class (extends Actor)	Class representing Pet Relationship: 1. Extends Actor to achieve actor functionalities 2. Has association with Behaviour to access Pet behaviours 3. Has dependency with Ability to add Ability.CAN_ENTER_FLOOR and Ability.IMMUNE to Pet.	
Created a LightningEel class (extends Pet)	Class representing LightningEel pet Relationship: 1. Extends Pet to achieve Pet functionalities.	

	2. Has dependency with FollowBehaviour and LightningAreaAttackBehaviour to add the behaviours 3. Has dependency with Status and Ability to gain access to Status.HOSTILE_TO_ENEMY, Status.CHOSEN_PET and Ability.IMMUNE.	behaviours while Pet abstract class focuses solely on implementing the general behaviours for all pets. 2. Open-closed Principle: When new pets are to be created, we only need to extend the Pet abstract class without modifying the existing classes such as LightningEel. This results in better flexibility and easier expansion of the program 3. Liskov Substitution Principle (LSP): ChoosePetAction class extends from Action abstract class. By doing this, we can substitute Action with ChoosePetAction without modifying the intended behaviour of the program. 4. Interface Segregation Principle: The child classes of Pet only extends Pet abstract class and only implements methods relevant to Pet. It solely manages pet-related logic without depending on any other irrelevant methods or interfaces. 5. Depedency Inversion Principle: High level modules such as the World class depends on the Actor abstract class which depends on the Pet abstract class. This avoids high level modules such as World class to have dependency on low level modules such as LightingEel, TankyKong and HealingUnicorn. Therefore, changes in the Pet subclasses will not affect the World class' intended functionality. 6. Don't Repeat Yourself Principle: Common pet behaviours such as the playTurn method are only implemented in Pet
Created a TankyKong class (extends Pet)	Class representing TankyKong pet Relationship: 1. Extends Pet to achieve Pet functionalities. 2. Has dependency with FollowBehaviour and TankAttackBehaviour to add the behaviours 3. Has dependency with Status and Ability to gain access to Status.HOSTILE_TO_ENEMY, Status.CHOSEN_PET and Ability.IMMUNE.	
Created a HealingUnicorn class (extends Pet)	Class representing Healing pet Relationship: 1. Extends Pet to achieve Pet functionalities. 2. Has dependency with FollowBehaviour and	

	HealingBehaviour to add the behaviours 3. Has dependency with Status to gain access to Status.HOSTILE_TO_ENEMY, Status.CHOSEN_PET.	abstract class. The subclasses such as LightingEel, TankyKong and HealingUnicorn will only need to inherit the behaviour from the Pet abstract class leading to prevention of code duplication. 7. Promotes Connascence of Meaning: The allowableActions method in each subclass checks if the pet has the Status.CHOSEN_PET capability to determine whether it should follow the actor and conduct its special behaviour. The meaning of Status.CHOSEN_PET is well-understood across the pet subclasses to indicate a selected pet. 8. Avoids design smells such as Large Class: The Pet abstract class does not contain all the logic for each pet' s functionality. Instead, respective behaviours are implemented in separate classes such as LightningEel, TankyKong and HealingUnicorn. This separation avoids having multiple functionalities in a single large class, reducing complexity. Disadvantages of Finalised Design Pattern: 1. As more pets are to be created in the future, it can lead to bloating of the program as the number of classes increases. 2. Creating multiple classes instead of implementing everything into a single class can make it harder to locate a specific class, especially as the total number of classes grows substantially.
Modified Application class to create new instance of LightningEel, TankyKong and HealingUnicorn to be placed on the Gravesite Map.	Class representing the Application Relationship: Has dependency with LightningEel, TankyKong and HealingUnicorn to create new instance.	