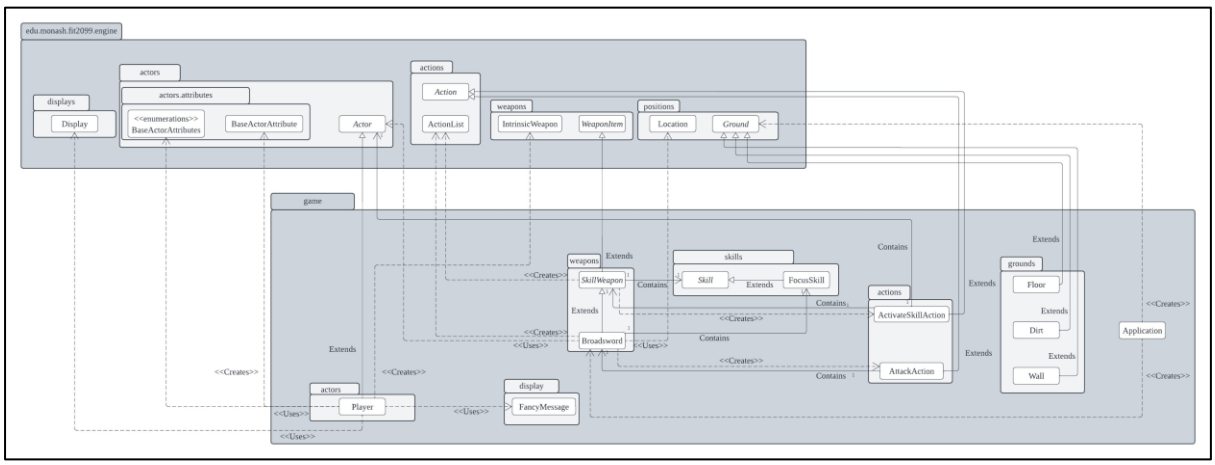
FIT2099 Assignment 1 Design Rationale

Requirement 1:



- A SkillWeapon class is created which is a sub-class of WeaponItem class. This class is solely for those weapons that have skill. Since there are weapons that may or may not have skill, introducing a SkillWeapon class can have some common method between all the weapons but also adding a new skill feature. Hence, this obeys the Single Responsibility Principle (SRP) as SkillWeapon class manages all the weapons that have skill only. This also follows the inheritance principle of OOP, which allows the code of the existing class to be reused and polymorphism. By inheriting from WeaponItem class, the SkillWeapon class can access or overrides some common attributes and methods which are shared by all weapons. This can help to avoid repetitions (DRY). SkillWeapon class is made abstract as different weapons may have different skills, and hence having different skill effect. By making SkillWeapon class abstract, it forces all the sub-classes to implement more specific details and prevent SkillWeapon class from being instantiated as it is incomplete. For those weapons which do not have any skill (if are introduced in the future), the weapons can directly be inherited from the WeaponItem class.

- Since all the skills will have same attributes such as skill duration, stamina cost, it is better to make an abstract Skill class which consist of all the common attributes. By doing this, the Skill class will only have one single responsibility which is focus on skill of weapon only. The Skill class basically serves as a blueprint for all the skills of the weapons by containing all the common attributes or methods such as getters which is shared by all the existing skill. The Skill class can be easily extended by using inheritance to maximize the reusability of codes. For example, FocusSkill class is extended from the Skill class and it inherits all the attributes and methods in the Skill class. However, it also needs to implement more specific details such as the actual value of stamina cost as this is not completed in the abstract parent class. Hence, it achieved polymorphism since the skill may have similar characteristics but different in the specific implementation. If there are some common attributes to be added into the skill features such as skill cooldown in the future, it can be simply added at the Skill class. This greatly enhances the efficiency of fixing the code or adding a new feature in the future.

- A Broadsword class is created by inheriting from the SkillWeapon class. This allows the Broadsword class to access the common attributes and methods shared among all the weapons and get to have methods such as activating skill and avoid repeating codes (DRY).

The SkillWeapon class will contain a Skill inside as it needs to access the attributes such as stamina cost and skill duration. Since different weapons may have the same skill, the skill is implemented as a class instead of interface so that the skill class will contains the basic attributes such as stamina cost and can provide to the SkillWeapon class if necessary. This strictly follows the SRP as SkillWeapon class only manages those weapon that have skill while the Skill class manages all the existing skill for the weapon.

- ActivateSkillAction class extends from the Action class. Similarly, this class tells the players what action they can perform and it inherits all the common methods shared by all the action class (DRY). The actual implementation is done by overriding the existing method in the parent class. Like the AttackAction class, this class manages its own responsibility, which is telling the user whether the activation of skill can be performed. This obeys the SRP as the SkillWeapon class manages the weapon that has skill, and it contains a Skill class which tells the specific details of a particular skill and it uses the ActivateSkillAction class to tell the player that a skill can be activated. Each of these classes managed their own responsibility, which will help to improve the readability as they do not have extra responsibility in their classes.

Pros:

- Allows for code to be reused and polymorphism by using inheritance.
- Reduce the complexity and increase readability by using abstraction and encapsulation.
- Can be easily extended if new skills or weapon (with or without skill) is added.
- Characteristics of every skill such as skill duration and stamina cost can be stored in their respective class, allowing the other weapon that uses the same skill to avoid repeating the characteristic of the skill.

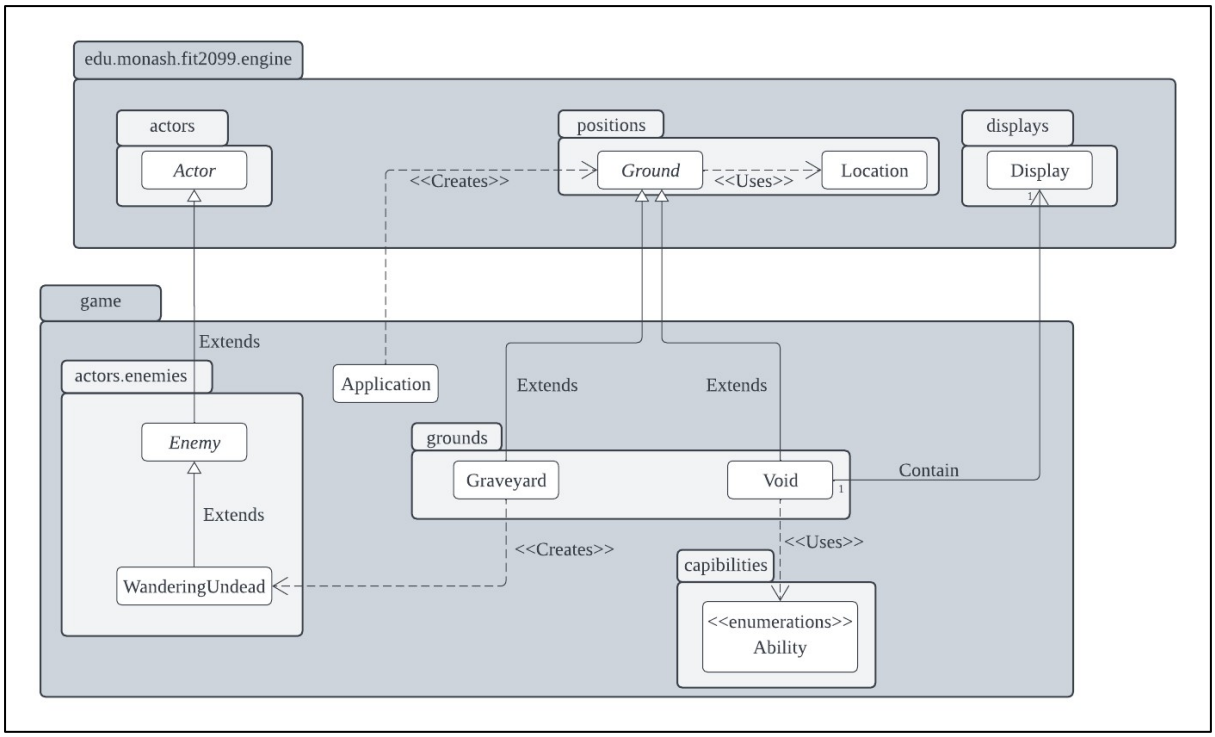
Cons:

- May cause over complexity by using too much inheritance or abstraction.
- May create tight coupling between classes. (Each skill weapon class is associated with a skill class and they use ActivateSkillAction and AttackAction class).
- Since the skill is not implemented as an interface, if a weapon has more than one skill, it would need to modify the weapon class to allow the player to determine which skill to be activated.

Future extension supports:

- Weapons with skill can be inherited from the SkillWeapon class like broadsword. Weapons without skill can directly be inherited from the WeaponItem class.
- All skills can be inherited from the abstract Skill class.
- If any features such as weapon with skill, skill of weapon and the activation of the skill needed to be changed, they can be modified in their respective class since the three classes are separated and each manages their own responsibility.

Requirement 2:



- The void and graveyard are set as sub-classes of Ground class as they are the basic element in the game map, and obviously, they are not items which may be consumable or portable. By using inheritance, this can help to reduce repetition (DRY) as they have similar characteristics and shared common attributes.

- By default, any actor that stepped on the void will be dead. So, any actor will be killed instantly if the void detects that there are player stepping on it unless they have the ability to immune to void. This ability can be set as a capability for the actor.

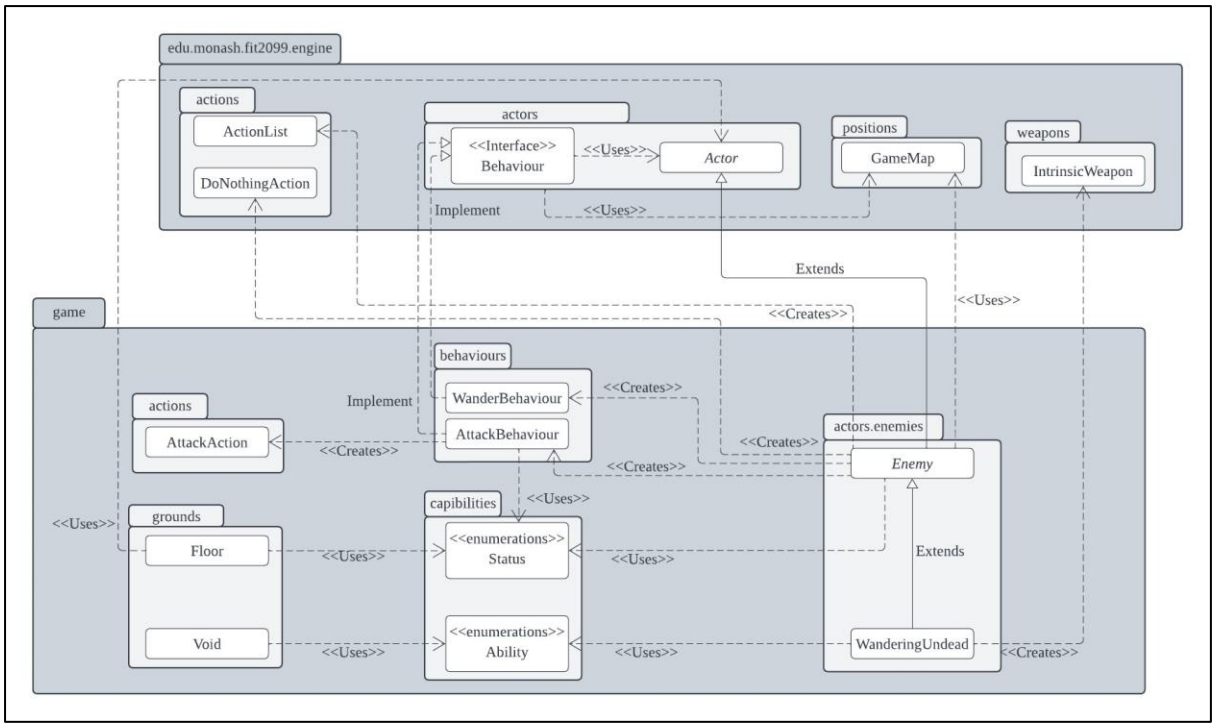
Pros:

- Reduce repetition, making the code to be less complex.
- For new actors that are immune to the void, simply add the ability to immune to void to the particular actor without modifying any existing class or method.

Future extension supports:

- Developers can utilize the ability or status of the actor in the game if there are specific grounds that may have special interactions with the actors. Override the existing method of the ground or implement a new interface to add the new game features.

Requirement 3:



- An AttackBehaviour class is added and it implements the Behaviour interface. By implementing an interface, the Behaviours class tells the AttackBehaviour class what should be completed in this class. This can help to achieve loose coupling. Similar to the WanderBehaviour class, AttackBehaviour class tells the wandering undead to attack the player as first priority while the WanderBehaviour class tells the wandering undead to wander around if there is no player nearby. The two classes both have similar properties which is to describe the behavior of the wandering undead but they each manage their own responsibility (SRP). This makes code easier to maintain and extend in the future.

- The wandering undead will have a danger status and the ability to immune to void. These status and ability are added as a capability to the actor so that the ground will only need to check whether they consist of any specific status or ability to determine the action that they can performed. This can help to improve the readability and reduce the complexity of the code for the Ground class (only need to check the capability of the actor instead of using if-else to specify which actor can perform some specific actions).

Pros:

- Every class will be focused on their own responsibilities and hence it will be easier to maintain.

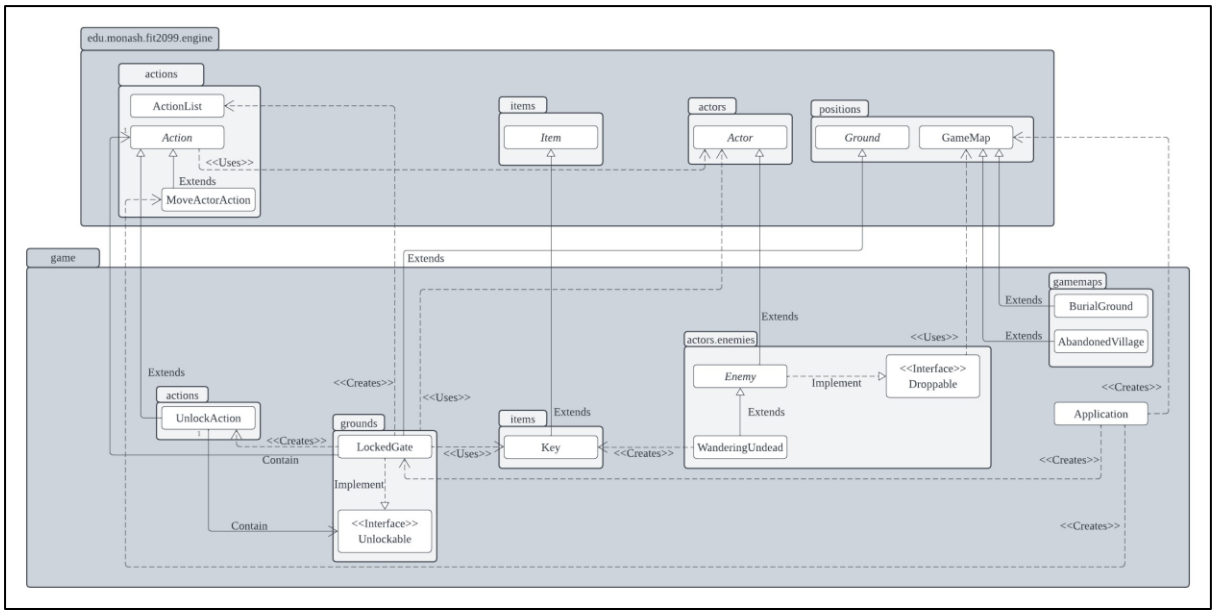
Cons:

- Creating too many classes or interface may cause the code to be more complex and worsen the readability.

Future extensions support:

- If there is an ambiguity in the status or ability, for example there are enemies that are danger but they still can enter the floor, the status or ability can be easily modified to be more detailed and specific for certain subjects.

Requirement 4:



- The burial ground and abandoned village map have been added as classes inherited from the GameMap class instead of creating manually in the Application driver class. This approach allows the gamemap class to manage their own responsibility. Since all attributes and methods for the gamemap are implemented in the GameMap class, it makes sense to inherit from the GameMap class for both the BurialGround and AbandonedVillage class. This can help to avoid repetition in code (DRY). Besides, by inheriting from the GameMap class, the Liskov Substitution principle can be achieved as all the method in the GameMap class is also existed in the BurialGround and AbandonedVillage class. Hence, different gamemap can be easily added into the game without needing to modify the driver class or other existing classes.

- LockedGate is implemented as a sub-class of the Ground class as it has some characteristics that are shared among all the Ground class, for example, the actor cannot enter in certain circumstances. However, locked gate has a special characteristic that is not used by the other ground type, which is the unlock feature. In this case, a new interface, Unlockable, is introduced and is implemented by the LockedGate class. Since for similar objects such as locked gate, chest or a wooden door (if added in the future), they will have an unlock method. Hence, by implementing an Unlockable interface, it forces those classes to implement the exact details on how they should be unlocked. By doing this, we also obey the Open/closed principle (OCP), where a new functionality is added without modifying the existing class.

- A key class is added by extending it from the Item class. The use of inheritance can help to reduce the repetition (DRY) as the Key class does not need to explicitly code out those common methods such as getDropAction or getPickUpAction. It also obeys LCP as all of the methods and attributes in it exists in the Item class.

- The LockedGate class will create an UnlockAction to tell the player that they can perform an unlock action to the gate. The UnlockAction is inherited from the Action class and the method is overridden for a more specific detail. This action will contain an unlockable object (in this case is the LockedGate itself which implements the Unlockable interface) to determine the things that are going to be unlocked. This follows the Dependency Inversion Principle (DIP)

where the UnlockAction depend on the Unlockable interface rather than detailed low level code. By doing this, as long as the interface does not change, this part of the system will not be affected if there are modifications in other classes.

- The LockedGate class will contain an attribute which stores the MoveActorAction. Since every gate that is added in the game may lead to a different location, it makes sense to contain a MoveActorAction which tells the player where they will be going if they enter the gate.

- The enemy in the game such as the wandering undead will be implementing the Droppable interface which implies that they may drop some item for the player after defeated. Since every enemy may have different item to drop, it makes sense to make the drop method abstract so that they can be completed with specific detail in the class that implements the interface. This will achieve the OCP as the original code does not change but instead an additional method is added to obtain a new feature. This can help in future extension as if any new functionality needs to be added, it can be implemented as an interface which will make the code easier to maintain and extend.

Pros:

- The code can be extended easily and easier to maintain as every class manages their own responsibilities (SRP).
- Droppable and Unlockable interface is introduced and can easily be used by those classes that have the characteristics. These interfaces are very small (consist of only one method) and hence it can achieve the Interface Segregation Principle (ISP) and also ensure that the classes that implement these interfaces can be more focus and tend to have a single responsibility and they have higher chance to fully substitute the interface.
- Since the map of the game is made into a class, the application driver class can be more clean and less complex as it does not needs to include the map made in arrays.

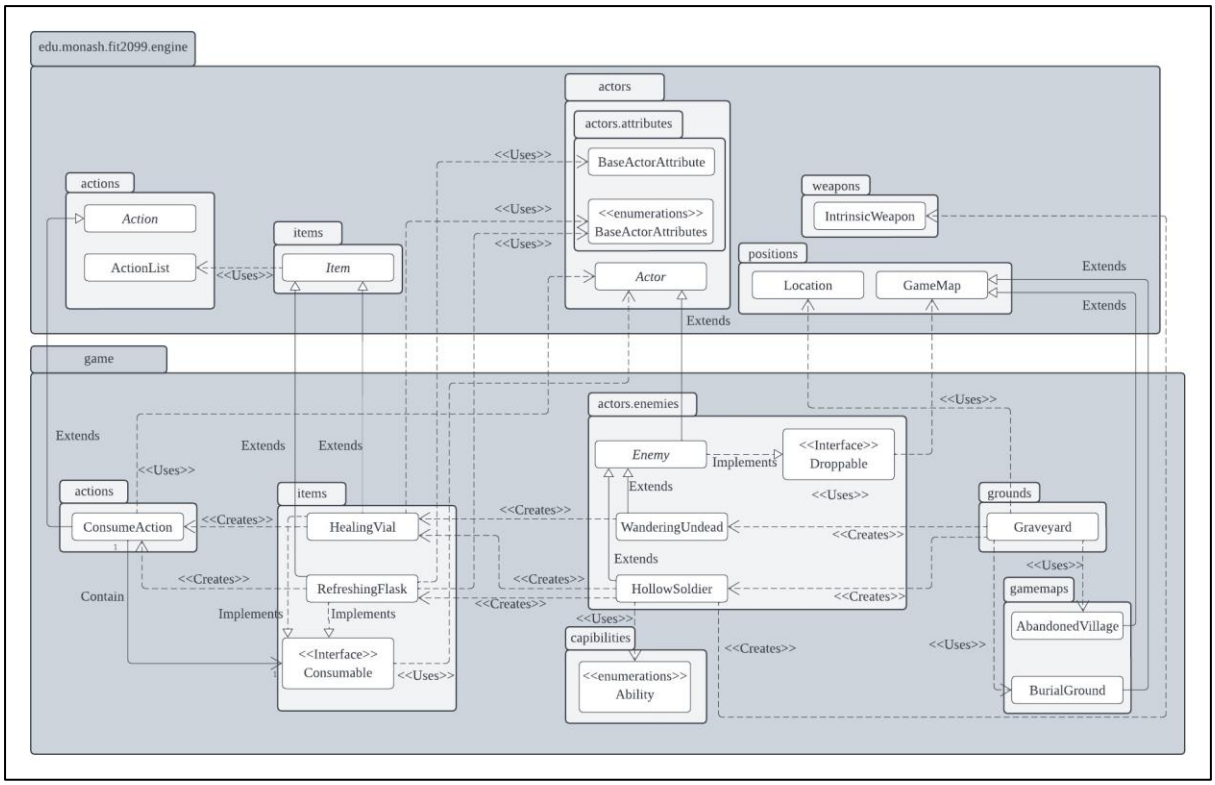
Cons:

- Too many interfaces or inheritance used may cause the code to be more complex and worsen the readability.

Future extensions support:

- Developers can easily modify the interface if there are any changes in the game feature as the design follows OCP and hence they are easier to maintain and extend.
- If new objects that can be unlocked are added, implements Unlockable interface for those particular classes.

Requirement 5:



- An Enemy abstract class is added and it is inherited from the actor class. This serves as a blueprint for all the enemies they may encounter by the player in the game. This class consists of the common attributes of all the enemies such as having the attack behaviour, wander behaviour, danger status, capability to drop item after being defeated, etc. By having this Enemy abstract class, every enemy can inherit from this class and get access to those common attributes and methods shared by the enemies. This can help to avoid repetition (DRY) by using inheritance. The Enemy class is also made abstract as it is not completed and should be instantiated. By using abstraction, it forces the sub-classes to implement all the specific details of the enemy. Since it is common for enemies to drop some item if they are defeated by the user, the Enemy class will implement the Droppable interface. The drop item method will then be completed in the sub class of the Enemy class so there will be specific details on what item may be dropped or the probability of dropping an item.

- Refreshing flask and healing vial inherits from the Item class. By using inheritance, we can avoid repeating (DRY) the common attributes and method shared between all the items. These two classes both implement Consumable interface as the items will be consumed after they are used. Since this functionality is not common for all the items in the game, they are only implemented for the specific class instead of implementing it in the Item class. By implementing the Consumable interface, this design can achieve OCP as it makes the code easier to maintain and extend and it does not require modifying the existing code. The two classes will also use ConsumeAction which is extends from the Action class. This helps to avoid repeating code (DRY) as all the actions have similar methods.

Pros:

- The code is easier to maintain and extend as OCP is achieved.
- Every class manages their own responsibilities (SRP)

Cons:

- In this design, all the enemy is assumed to be dropping an item if defeated by the player and attack the player if they are nearby. Hence, if there is an enemy that violates these rules, the Enemy abstract class might need some modification and add those features separately to the respective enemy instead of adding in the blueprint of all the enemies.

Future extensions support:

- Developers can add new behavior or implement new interface in the abstract Enemy class if the behavior or capabilities are common for all types of the enemy in the game.
- For items that can only used for certain times, Consumable interface can be implemented by the particular item.