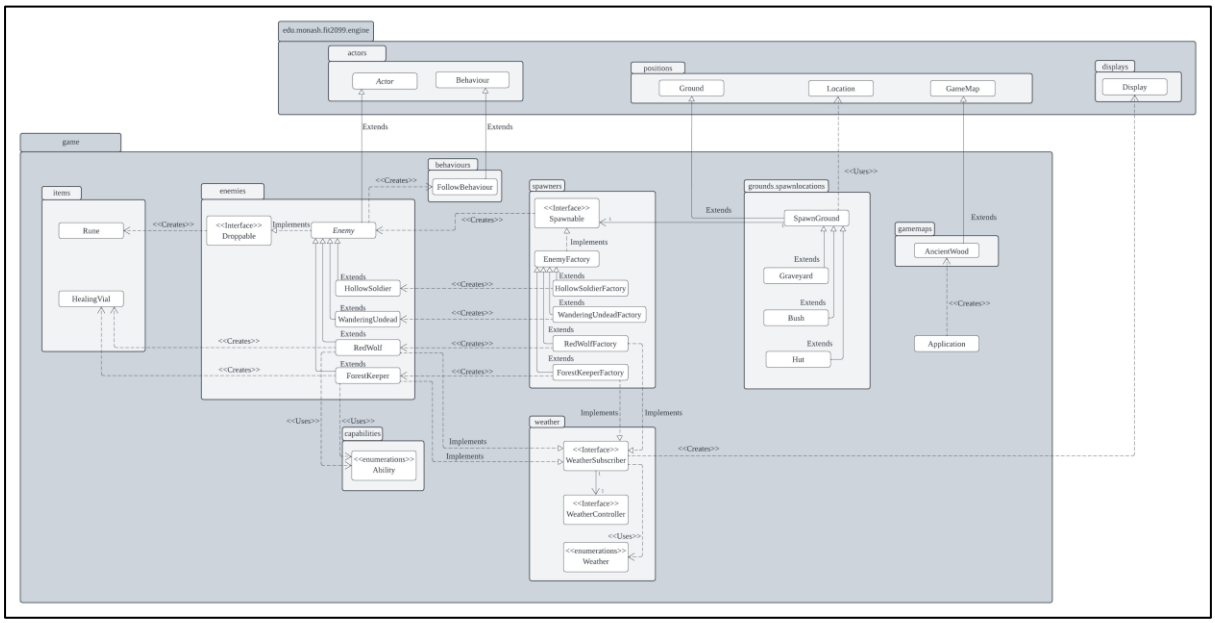
FIT2099 Assignment 2 Design Rationale

Requirement 1:



- In this requirement, we had changed the spawning mechanism which we kind of use instanceof method to spawn a valid enemy at a valid map in the last assignment. In this assignment, we introduce a spawnable interface which contains a spawn enemy method in it. Then, an abstract EnemyFactory class implements the Spawnable interface. This EnemyFactory class can be extended to form a concrete enemy factory class to form a particular type of enemy. This helps to avoid code repetition (DRY). The EnemyFactory class acts like a spawner which basically will manage the responsibility to spawn an enemy on the ground.

- We also introduced a SpawnGorund abstract class which extends from the ground. This SpawnGorund class is the base class for those ground that can spawn enemies. In the SpawnGorund, it will contain a Spawnable instance which will help to determine the type of enemy to be spawned. This follows SRP principle as now every class only focuses on their one and only responsibility and hence the code will be easier to manage and maintain in the future. This also follows the OCP principles as for a new enemy introduced, we can simply add a new enemyfactory class. Then, for different ground instances of the same type, we can also let them spawn different enemies by passing the respective enemyfactory to the constructor.

- The red wolf and forest keeper also implements droppable interface as they may drop some items after being defeated by the player. This helps to ensure that the drop implementation is done for every concrete class.

- Since the requirement stated that the enemies in ancient wood will have follow behaviour, all the enemies from the ancient wood is added with a follow ability in the constructor. The following behaviour will be added to the enemy when the enemy has a follow ability and it detects an actor nearby, which has the hostile to enemy status. By doing so, it follows OCP principles as we can simply let an enemy have the follow behaviour by adding the ability to them.

Pros:

- Follows SRP. A spawner is introduced which serves for the purpose of spawning an enemy. If compared to assignment 1, the code now will be much easier to maintain as cohesion is achieved.
- Easier to extend without needing to modify the existing code.

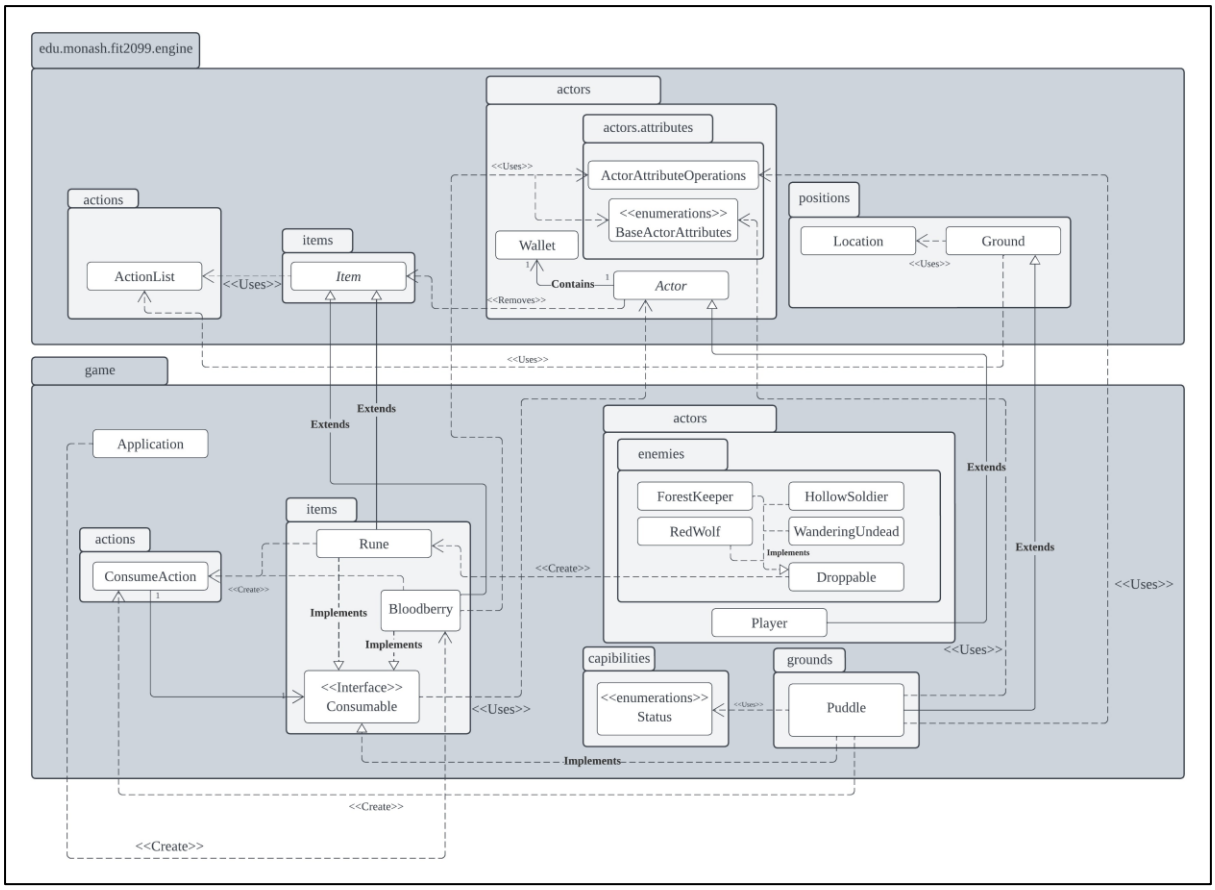
Cons:

- There will be an extra association between ground and spawnable instance, which may cause a tighter coupling.

Future extensions support:

- If a new enemy is introduced, simply create a new spawner class for the enemy.
- A spawning ground can now choose to spawn different enemies by passing the respective enemyfactory instances into the ground constructor.
- For a new enemy or existing enemies which decide to have a follow ability, simply add the follow ability to the enemy's capability set, and it will then start to follow a player when a nearby player is detected.

Requirement 2:



- Rune class extends the Item class and implements the Consumable interface. It implements the consumable interface because it can be consumed by the player to increase the player's balance. By extending to the Item class and implementing the Consumable interface, we follow the principle of DRY or Don't Repeat Yourself, since all the functionalities of Runes follow the functionalities of Item class and Consumable and it will enforce the Rune class to implement the consume method so that it can be used by the ConsumeAction class.

- Bloodberry class is extending to the Item class for the same reason as Runes, it can be picked up, and dropped. This item also implements the consumable interface as it can be consumed and increase the player's maximum health. This helps to achieve Open Closed Principle as we only extend the code without modifying the existing code.

- An extra constructor is added in the ConsumeAction class. This constructor accepts an additional argument, description. This extra argument is used in the menuDescription for certain classes that implement the Consumable interface such as Puddle. The reason is that the default string returned by the menuDescription method is designed for item type only. So, for a Puddle which is a ground type is not suitable to use the default string and hence they should have their own description. By doing this, the code will be more extensible which follows the OCP principle.

- For the Puddle class, it extends to the Ground class as it's a type of ground. It also implements the Consumable Interface because it can be consumed to increase the player's maximum health.

Pros:

- Reusability: By creating a second constructor in ConsumeAction class, we can now handle different types of consumables (non-item type) with different descriptions without modifying the ConsumeAction class anymore. This adheres to the Open-Closed Principle.

Cons:

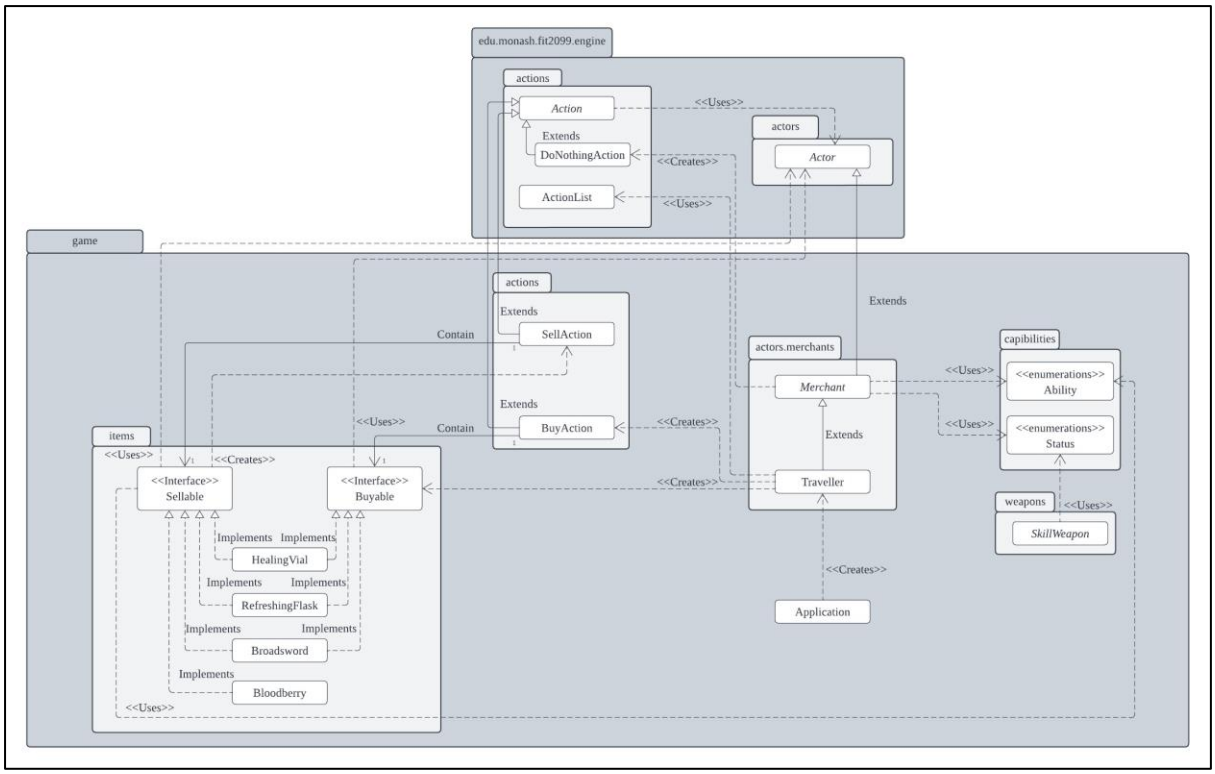
- None for now.

Future extensions support:

- If there is an item added in the future, as long as it can be consumed by the player either by drinking or eating or similar, they can use the consumeAction to perform the action. This includes non-item type such as a puddle which belongs to ground. The second constructor for the consumeAction class allows the developers to pass in the menu description of the consumable entity as an argument which will be used to display on the console menu for the player to choose what action they want to perform.

- Runes, and Bloodberries was simply implemented by only extending the classes we had before, and by adding a second constructor in the ConsumeAction class, we have strongly built a foundation for some of the SOLID principles such as the Open-Closed Principle. Which means in the future, if we have more consumable items to add, we can just extend without modifying the ConsumeAction class at all.

Requirement 3:



- Traveler class is extended from a Merchant abstract class which is extended from Actor class. Merchant abstract class acts as a parent class for all the actors that can buy or sell items to the player. Having a Merchant class allows us to prevent code repetition (DRY) since all the common characteristics of the merchants can be added into the Merchant abstract class so the subclass that inherits from the Merchant class will also have this characteristic.

- All the merchants will have a neutral status as they cannot be attacked by any of the actor in the game. The merchant will also have a trading ability so that if a player encounters an actor with this ability, they can sell items to them.

- Traveler can be added as a subclass of Ground with the following reasons:
  - Traveler does not need playturn method as it cannot move around. They also cannot attack or being attacked by other players, which fulfill the characteristic of a ground type.
  - Traveler should be able to sell items to the player. This can be simply done by returning a buyAction when the allowableAction method in the ground class is called.
  - Travelers does not need a HP. They don't necessarily need a wallet and item inventory since they can sell an item infinite number of times to the player and they have infinite amount of cash. Besides, if they bought an item from the player and the item is not sellable, it will be useless to add the item in the inventory.

  However, in our design, traveler is added as a child class of the Actor because it is possible that a wandering traveler is introduced to the game in the future. In this case, if the Traveler is implemented as a subclass of Ground, we might need to modify the whole class since a Ground is not allowed to move. This will violate OCP principle and will cause us a lot of time to modify the code. Hence, the Traveller class is implemented as a child class of the Actor.

- Buyable and sellable interfaces are introduced. Having two separate interfaces for this feature follows ISP principle as some items may be sellable but they are not buyable. If the two interfaces are combined into one, then we might need to provide a dummy implementation if we encounter the case we mentioned above where some items may only support one function but not two.

- Sellable item will have a base selling price as an attribute in its class. This is because the price of every item sold by the player to the merchant is fixed. However, the price of the items sold by the merchant may be different. Hence, the constructor of the buyAction will accept an extra argument, price, which allows the merchant to declare their own price for the items they sold.

- BuyAction class is referenced to a Buyable interface and SellAction class is referenced to a Sellable interface. This allows the BuyAction and SellAction class does not tightly coupled to a particular item. This follows OCP and DIP principle as if there is a new item which is buyable added into the game in the future, we simply just need to implement Buyable interface to the item and it can then be bought from the player without needing to modify any of our existing class. Having a BuyAction and SellAction class separately also follows the SRP principle as each of the class manage only one responsibility of their own.

Pros:

- This design follows OCP principle as if there is a new item or actor added, we do not need to modify the existing code but simply just extends from it.

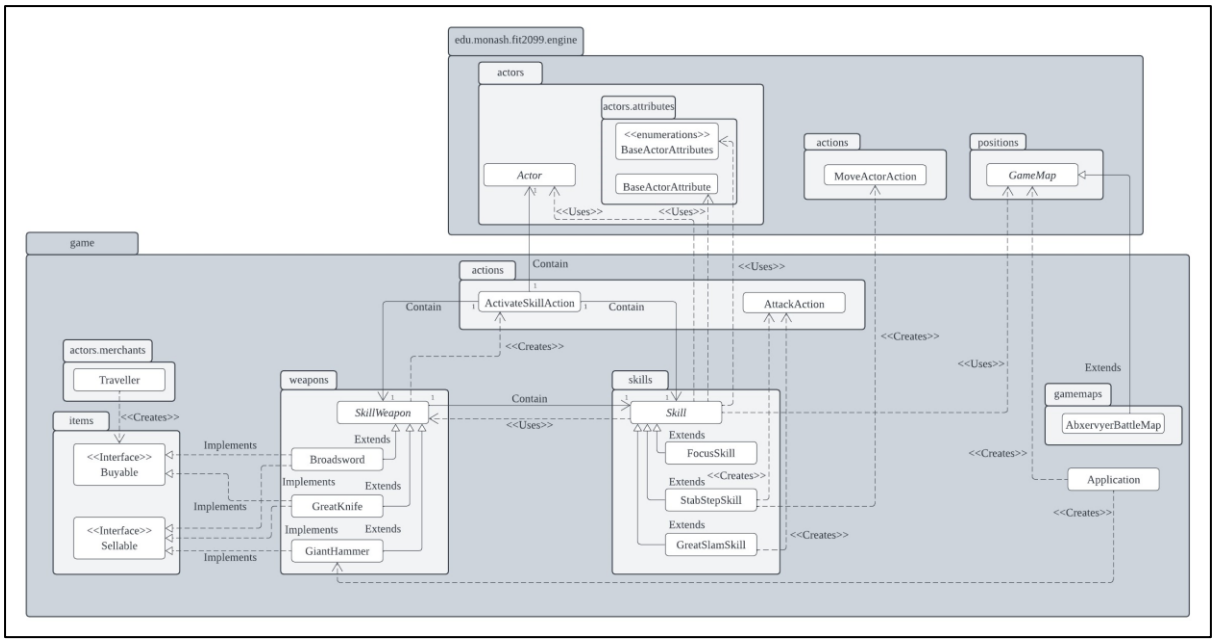
Cons:

- Traveler class implements as a subclass of Actor has a few redundant features, for example, the health and stamina (they cannot attack or being attacked), the playturn method (basically just returning a new DoNothingAction), the wallet and iteminventory (they have infinite supply of item and cash)
- Since every traveler has different items to sell and the item is sold at different prices, the allowableAction for every Traveler might need to be manually coded.

Future extensions support:

- If a merchant type actor is added to the game and they can move around, simply override the playturn method and add the wander behaviour to the concrete subclass of the merchant class. In our design, we are assuming most, if not all, of the merchant type cannot move around.
- If a new buyable or sellable item is added into the game, simply implements the respective interface to the item.
- For a new merchant type actor, specified the item that is sold by the merchant and the respective price in the allowableAction method.

Requirement 4:



- The implementation of the skill feature has changed as compared to assignment 1. In this design, the effect of the skill is managed by the skill class instead of managing by the weapon that contains the particular skill. This follows the SRP principle as now the weapon class manages all the actions related to the weapon itself while the skill class manages the effect of the skill on the weapon that activates the skill.

- The new StabStepSkill and GreatSlamSkill are inherited from the skill class. The reason to implement it as a class instead of an interface is that all the skill will have some common attributes such as stamina cost or duration. Having an abstract Skill class can help to avoid code repetition (DRY).

- The ActivateSkillAction class now contains an additional attribute, skill. This additional skill attribute allows the player to choose what skill to be activated. This follows OCP principles as if a weapon with multiple different skills is added into the game, the weapon can simply create multiple ActivateSkillAction classes which contains different skill for the player to choose. In design for assignment 1, it is assumed that the weapon only allows to have one skill. Hence, if multiple skills weapons are added in the future, the existing code has to be modified.

- The ActivateSkillAction class consists of two constructors with one having an Actor type target. The purpose of this constructor is to allow the target to be specified for skills such as Stab Step and Giant Slam which is used to attack the enemy directly.

- GreatKnife and GiantHammer are extended from the SkillWeapon class as they both have skill to be performed. Through inheritance, repetition of code can be avoided (DRY).

Pros:

- Every class managing their own responsibility (SRP) can help to make the code easier to maintain.
- Using inheritance avoids the repetition of code.

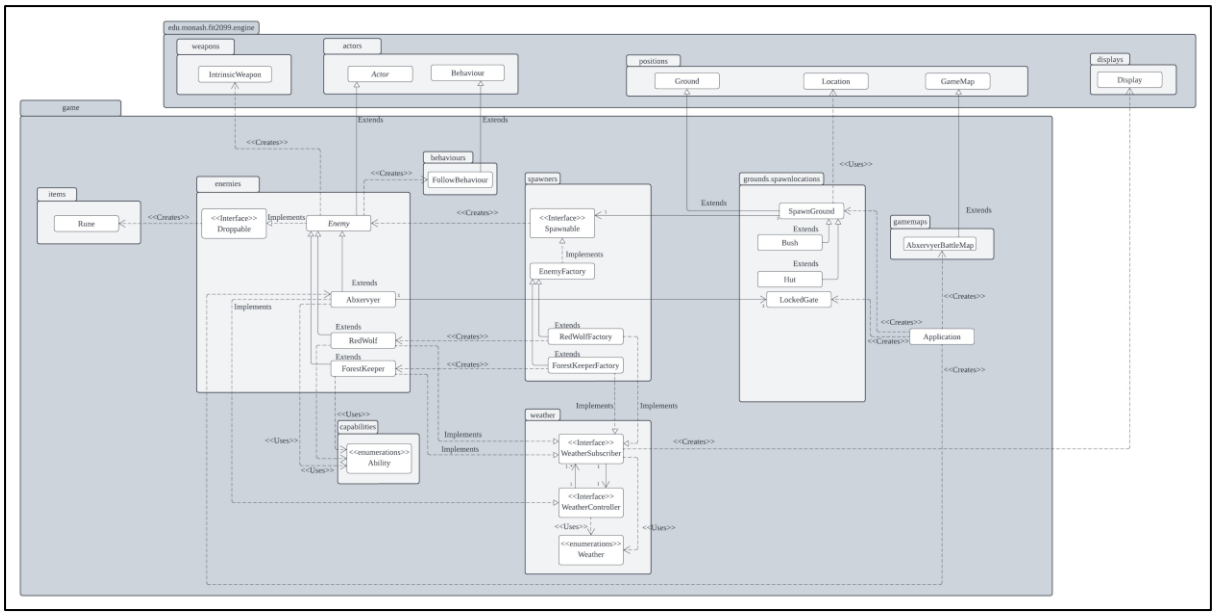
Cons:

- There are too many associations and dependencies between SkillWeapon class, Skill class and ActivateSkillAction class which may cause our classes to be tightly coupled.

Future extensions support:

- If new weapons with multiple skills are introduced, simply add the skills as an attribute in the weapon class and pass the skill to ActivateSkillAction class. The skill will manage the effect on the weapon itself.
- New skills introduced can be extended from the Skill abstract class to avoid code repetition.

Requirement 5:



- The boss is added by extending from the Enemy class which implements the Droppable interface. This helps to ensure that the drop method will be implemented in the boss class as it may drop items after being defeated by the players. By extending from the Enemy class, we can avoid code repetition (SRP).

- Since the boss can control the weather of the map, instead of containing a map in the boss class (which may help to manage the effect of weather for the spawning location and the actor), we introduce two new interfaces, WeatherController and WeatherSubscriber. WeatherController basically represents a remote control for the weather and it can manage those that will be affected by the weather, and hence it is implemented by the boss. Meanwhile, WeatherSubscriber is implemented by those game entity that may be affected by the weather. By separating the two interfaces, we can follow ISP as each interface consists of methods that are only specific to the responsibility they managed.

- By using the two interfaces, we can also achieve OCP principles as it is extensible without needing to modify the code. If there is a new enemy that can control the weather, they can implement WeatherController while if there is a new game entity that is affected by the weather, they can implement WeatherSubscriber.

- The enemies that implement WeatherController will consist of a list of game entities that implement WeatherSubscriber. By having this association, this makes the WeatherController to be easier to notify the weather to the subscriber and call the weather effect to act. For those WeatherSubscriber, they may have reacted differently for different weather. By implementing it as an interface, it enforces the implementation of the weather effect to be done in the class that implements it. This also helps to achieve SRP as the WeatherController is responsible to control the weather and report the current weather only while the WeatherSubscriber will do the action based on the weather effect.

Pros:

- Achieve OCP as it is easier to extend without needing to modify the existing code.
- Each class manages their own responsibilities causing a high cohesion and hence will be easier to manage and maintain.

Cons:

- There is only two weathers for now and the weather is implemented as an Enum. The weather subscriber uses if-else to check the weather and apply the respective effect. So, if there is a new weather introduced, the effect of the weather might need to be explicitly coded out in every class that implements WeatherSubscriber.

Future extensions support:

- Implements WeatherController for new enemy that can control the weather. The details implementation will be forced to complete in the class as well.
- Implements WeatherSubscriber for game entity that is affected by weather. However, the effect of the weather on the game entity needs to be coded out explicitly as different game entity may react differently to the same weather.
- For a new weather introduced, add the weather into the Weather Enum.