Requirement 1 (Raynen)

To fulfill Requirement1 a SpawnableEnemy abstract class was made which could be extended by enemies that are “Spawned”, a SpawnableGround abstract class was made which could be extended by Grounds that spawn enemies, and a Follow interface was made which could be implemented by enemies that create an FollowBehaviour instance. An Enemy abstract class was also made which SpawnableEnemy and all enemies in the game would extend.

Enemy:

The Enemy abstract class which extends Actor and implements DropAction should be extended by all Enemy actors, for instance the RedWolf, ForestKeeper and ForestWatcher. The Enemy abstract class consists of the abstract method dropItem and the overridden methods unconscious(), playTurn() and allowableActions(). It also consists of a TreeMap called behaviours consisting of integers and Behaviour instances which are arranged in order in the TreeMap.

Why:

This strategy prevents the repetition of code as methods which all “enemies” should have can be found in the Enemy abstract class and can be used by the classes that extend Enemy. Regarding future extensions, if “Enemies” were to have more similar methods or variables, they could simply be added into the Enemy abstract class instead of every class which extends Enemy and if any of the methods in Enemy were to differ in implementation in the classes which extend them, the methods could just be overridden. This allows us to have different implementations for certain “Enemies” while still using methods from Enemy that have not been overridden and are not abstract.

Pros:

- Maintainability is enhanced as if a specific method or variable should be changed in all Enemy Actors that have the same method implementation or variable value, the method or variable could just be changed in the Enemy abstract class rather than all of the subclasses.
- Adhering to DRY as code repetition is avoided by not repeating similar methods and variables in all classes that extend Enemy.
- Classes that extend Enemy are clear and concise as they only contain the constructor, overridden methods and implementation of abstract methods

- The Enemy abstract class takes the responsibility of having the methods that all “Enemies” should share in common unless overridden, which then leaves the classes that extend Enemy to contain the implementation of abstract methods and methods that are unique to those classes (Adhering to the Single Responsibility Principle)

Cons:

- The abstract method dropItem() assumes that all Enemies drop an Item when they die. However, if we were to have an Enemy that does not drop an Item when they die, the dropItem() method of that Enemy would have a do-nothing implementation of dropItem which is a LSP violation.
    - However, since our current program allows every enemy to drop an item, we feel that it is good implementation to have the dropItem() abstract method

SpawnableEnemy:

The SpawnableEnemy abstract class which extends Enemy and implements SpawnActor is extended by Actors that are spawned by Grounds, for instance the RedWolf being spawned by the Bush. The SpawnableEnemy abstract class consists of the methods spawn() (since it implements the SpawnActor interface) and newInstance(), an abstract method.

Why:

This strategy prevents the repetition of code as methods which “spawned” actors should have, are shared in the SpawnableEnemy abstract class and the spawning of the enemies themselves occur in the SpawnableEnemy abstract class. Regarding future extensions, if “spawned” actors were to have more similar methods or variables, they could simply be added into the SpawnableEnemy abstract class instead of every class which extends SpawnableEnemy and if any of the methods in SpawnableEnemy were to differ in implementation in the classes which extend them, the methods could just be overridden. This allows us to have different implementations for certain “spawned” actors while still using methods from SpawnableEnemy that have not been overridden and are not abstract.

Pros:

- Maintainability is enhanced as if a specific method or variable should be changed in all “spawned” Enemies that have the same method implementation or variable value, the method or variable could just be

changed in the SpawnableEnemy abstract class rather than all of the subclasses.

- Adhering to DRY as code repetition is avoided by not repeating similar methods and variables in all classes that extend SpawnableEnemy.
- Classes that extend SpawnableEnemy are clear and concise as they only contain the constructor, overridden methods and implementation of abstract methods
- For future extension, SpawnActor interface is implemented as we might have a different class that has spawning capability in the future. For the class to have the same functionality, we implement the SpawnActor interface for it to be able to spawn when required.
- The SpawnableEnemy abstract class takes the responsibility of having the methods that all “spawned” actors should have (only spawn() so far) which then leaves the classes that extend SpawnableEnemy to contain the implementation of abstract methods and methods that are unique to those classes (Adhering to the Single Responsibility Principle)

Cons:

- Currently in our implementation if there are Actors who are not Enemies that could spawn, they cannot extend the SpawnableEnemy abstract class as the class extends Enemy. This may result in duplicate code as another Spawnable<<Actor>> class would need to be created to cater for Actors that could be spawned who are not Enemies.

SpawnableGround:

The SpawnableGround abstract class which extends Ground is extended by Ground’s that spawn actors, for instance, the Bush spawning the RedWolf. The SpawnableGround abstract class consists of a final variable that holds the DEFAULT_SPAWN_MULTIPLIER (a double holding 1.0) and methods updateSpawnMultiplier(), resetSpawnMultiplier() and getPercentageOfSpawning() which are methods that could be similar for all classes that extend SpawnableGround. However, if the methods are different, they can be overridden as implementations could differ. The overridden tick() method in SpawnableGround spawn the Actors that are to be spawned.

Why:

This strategy prevents the repetition of code as methods that Grounds which spawns actors should have, are shared in the SpawnableGround's abstract class and the spawning of the enemies themselves occur in the SpawnableGround abstract class. Regarding future extensions, if Grounds that spawn actors were to have more similar methods or variables, they could simply be added into the SpawnableGround abstract class instead of every class which extends SpawnableGround and if any of the methods in SpawnableGround were to differ in implementation in the classes which extend them, the methods could just be overridden. This allows us to have different implementations for certain Ground types while still using methods from SpawnableGround that have not been overridden.

Pros:

- Maintainability is enhanced as if a specific method or variable should be changed in all "spawnable" Grounds that have the same method implementation or variable value, the method or variable could just be changed in the SpawnableGround abstract class rather than all of the subclasses.

- Adhering to DRY as code repetition is avoided by not repeating similar methods and variables in all classes that extend SpawnableGround.

- Classes that extend SpawnableGround are clear and concise as they only contain the constructor, overridden methods, implementation of abstract methods(None in SpawnableGround) and methods specific to that class

- The SpawnableGround abstract class takes the responsibility of having the methods that all Grounds which spawn actors should have and spawning of the enemies which then leave the classes that extend SpawnableGround to only contain methods that are unique to those classes (Adhering to the Single Responsibility Principle)

Cons:

- If the Game becomes increasingly complex, modifying the SpawnableGround abstract class to cater to the game's complexity by adding and modifying existing methods may become challenging. However, this trade-off is preferable to overriding and creating new methods in all subclasses which is harder to correct and maintain.

Follow:

A Follow interface was created which consists of the method followBehaviour(), Follow could be implemented by Actor's (Enemies) which will follow a specific Actor to implement the method followBehaviour() which creates a FollowBehaviour() instance.

Why:

The Follow interface was introduced as a way to enforce certain Actor's who would follow another actor to implement the method followBehaviour(), a separate interface was created as not all Actor's (enemies) in our game would have followBehaviour in their behaviours TreeMap. Regarding future extensions, the interface can be used by any Actor who wants to follow another actor and if Actor's who follow another actor would contain more compulsory methods other than followBehaviour() they can be added into the Follow interface. The Follow interface also serves as a way to encapsulate and abstract the FollowBehaviour, allowing me to implement different follow behaviour strategies for classes that implement Follow.

Pros:

- The use of Follow interface adheres to the Open-Closed Principle as a new interface was made which could be implemented by specific Actors rather than all Actors, as not all Actors would need to create a FollowBehaviour() instance.
- LSP is also not violated as this prevents certain Actors from having do-nothing implementations of the followBehaviour() method, which would be the case if this method were to be made as an abstract method in the SpawnableEnemy abstract class.
- Adheres to the interface segregation principle as there wasn't a Behaviour interface which consisted of methods followBehaviour, wanderBehaviour and attackBehaviour as if certain Actors were to not have a FollowBehaviour this would mean that the body of followBehaviour would be empty in the actor's class. A Behaviour interface consisting of followBehaviour, wanderBehaviour and attackBehaviour would cause certain Actors to have empty implementations of methods from the Behaviour interface. Rather a smaller interface Follow was made which could be implemented by Actors who would have the FollowBehaviour, preventing the existence of empty implementations in the actor's class.

Cons:

- This strategy allows for flexibility, but because equivalent behaviours must be implemented independently for each enemy class that implements Follow, it may result in some code duplication in the short run. However, the benefits of modularity and adaptability typically outweigh this trade-off in code duplication, as the modularity and encapsulation are maintained.

BehaviourOrder:

The BehaviourOrder interface was created to enforce certain classes (Classes that implement BehaviourOrder: AttackingBehaviour, FollowBehaviour and WanderBehaviour) to have a method (behaviourOrderRank()) which returns an integer that determines the priority of that Behaviour when adding a Behaviour into an Enemy's TreeMap of Behaviours. Ultimately this was done to reduce the use of Magic numbers (numbers without meaning) when adding a Behaviour in an Enemy's TreeMap of Behaviours as instead of hardcoding a number in the TreeMap's put() method, we use <<the behaviour instance>>.behaviourOrderRank() instead (Provides meaning to our code). However, having classes that stores the priority of their Behaviours (Classes that implement BehaviourOrder: AttackingBehaviour, FollowBehaviour and WanderBehaviour) is also good coding practice as it allows us to minimize the side effects of Connascence of Execution as the order in which the Behaviour is implemented is decided by the Behaviours upon execution and the coders do not have to remember the priority of Behaviours when adding a Behaviour into the TreeMap as calling the method behaviourOrderRank() for that specific Behaviour provides the priority. Moreover, maintainability is enhanced as when we need to change a priority value, we only need to change it in one place (Classes that implement BehaviourOrder: AttackingBehaviour, FollowBehaviour and WanderBehaviour) and not in every occurrence of the priority value used (Methods that add Behaviour instances: ex:FollowBehaviour) if using magic numbers, in the codebase (ChatGPT, n.d.).

Requirement 2 (Choong Lee Ann)

2099 A2 Requirement 2 Design Rationale

In requirement 2, we are required to implement currency in the games which are Runes dropped by enemies once defeated by the player. In order to implement this, we created a Runes class which extends Item. Since the runes need to be consumed before the money can be credited to the players wallet balance, we also created a ConsumeRunesAction class which extends Action. The Runes class will allow the ConsumeRunesAction and the ConsumeRunesAction will add the currency to the players balance and remove the Runes from the players inventory once it has been consumed. For the dropping of Runes by the enemies, we added it into the existing dropItem method in each of the enemy classes.

- Pros:
    - o By separating Runes and ConsumeRunesAction into two classes, we are adhering to Single Responsibility Principle where the Runes itself does not need to handle everything on its own, instead, Runes will only have the number of runes whereas ConsumeRunesAction will handle the adding of the player balance. This also makes it adhere to the Open Closed Principle.
    - o Adding Runes into dropItem will adhere to DRY principle as the method is similar to when enemies drop Refreshing Flask, Healing Vial and Old Key. Doing so also allows us to control the amount of runes an enemy can drop as the amount of runes may vary from different enemies.
- Cons:
    - o Based on our current implementation, if there were another form of currency other than Runes that needs to be implemented, they cannot share the same ConsumeRunesAction as the class was created specifically for Runes only. This may result in duplicate code if we need to implement a new form of currency in the future.

Furthermore, we implemented the ability for players to drink from puddles in order to regain health and stamina. In order to do this, we added an Ability, DRINK_WATER_FROM_GROUND so only actors with this ability can drink from the puddle. We also created a DrinkWaterAction class which extends Action to update the health and stamina of the player once the puddle has been consumed. DrinkWaterAction will call upon the UpdateStamina class to update the stamina and use the heal() method in Actor to update the health. We added this Action into the allowableActions method in Puddle class so if the player has the drink water ability and is standing on the puddle, they may perform the DrinkWaterAction. The previous implementations of HealAction and RefreshAction were not used in this implementation as they both are for Items that the players pick up and keep in their inventory whereas DrinkWaterAction is performed directly from the Ground.

- Pros:
    - Having the DRINK_WATER_FROM_GROUND Ability will ensure only players can drink from the ground so other entities cannot also drink from the puddle
    - By having a DrinkWaterAction, we adhere to SRP where the Puddle class does not have to handle its own instances of drinking and the DrinkWaterAction class will handle it instead.
    - Having an UpdateStamina class helps us to adhere to DRY principle where all instances of needing to update the stamina of the player can be done in one class and we do not need to repeat the code.
- Cons:
    - Having to create a new class to handle increase of health and stamina although previously there were methods defined to help increase the players health and stamina. However, since the DrinkWaterAction is to be used on Puddle which is not part of the Item class but it is the Ground class, so we cannot use the existing HealAction and RefreshAction as they are to be used on Items.

Lastly, we implemented a bloodberry which allows players maximum health to be increased once they consumed it. To do this, we created a Bloodberry class which extends Item. We also created an EatBerryAction class which will be called in the Bloodberry class similar to how Runes work. When the player owns the bloodberry, the EatBerryAction will be allowed which then lets the player consume the berry to increase their maximum health and then remove the berry from the players inventory once consumed.

- Pros:
    - Similar to the implementation of Runes, having separate classes for Bloodberry and EatBerryAction adheres to SRP. Furthermore, EatBerryAction is extensible for future implementations of other Items which could be consumed by the Player to increase their maximum health.
- Cons:
    - The current implementation of EatBerryAction may have some lines of code which are more specific to Bloodberry which would need to be changed if we wanted to use EatBerryAction for other Items (i.e., the name of the class is specific to Bloodberry)

Requirement 3 (Sarviin Hari)

1. For the implementation of the IsolatedTraveller selling an item to a player (playerpurchases), we have implemented an interface called PlayerPurchase which will be implemented by the items which are sellable by Isolated Traveller to Player (Player buys). The implementation is done by storing
    a. Pros
        i. This allows for a flexible way to define tradeable items and their prices. We can easily add or remove items and adjust their prices without modifying the code structure significantly as we abide by the SingleResponsibilityPrinciple where the item determines the buying price, the IsolatedTraveller determines the item that it can sell to Player and the Purchase Action class conducts the purchase between the Player and the Isolated Traveller based on specified conditions
        ii. Future extension
            - The PlayerPurchase interface is used to abstract the purchasing behavior of items from the trader. Any class that implements this interface can be added to the tradeableItems map, allowing for different trader types. This is certainly a better approach compared to having multiple interfaces for each item implemented by the IsolatedTraveller to determine the price of an item which is not suitable for extension and also goes against the Open-Closed Principle as this might cause an issue that we have to have multiple if… else or instanceof checks
            - The use of a map allows efficient retrieval of item prices based on the item, ensuring quick access to pricing information. For the a new Trader class (i.e. WanderingTrader), we can just customize the tradeableItems map to have different items and prices specific to that trader.
                a. This allows abstraction as only the items that are implemented by PlayerPurchase interface can be added to the IsolatedTraveller which allows encapsulation as the IsolatedTraveller controls the items and its standard price it can sell to player
                b. For this, we dont have to change the implementation of the ‘PlayerPurchase’ interface as it gets the price based on the standard price assuming every Trader have the same pricing strategy

    b. Cons
        i. It assumes that each item can be associated with a single price. If the game requires dynamic pricing based on various factors, this simplistic approach may not be sufficient.
            - However, since the requirement mentions that the the ‘same item will have a fixed selling price across different instances of traders’, this implementation is sufficient due to the standard selling price of the item across each instances of traders

- In addition, since the IsolatedTraveller holds the responsibility on the Standard Price of the Item, we can still dynamically determine the pricing strategy through methods in Isolated Traveller

2. For the implementation of the Player selling an item to an Isolated Traveller (PlayerSell), I used the implementation of capability of Enum type by adding the Status.MERCHANT to Isolated Traveller to indicate that the IsolatedTraveller can purchase an item and adding the ProductsAccepted.Bloodberry, ProductsAccepted.HealingVial, ProductsAccepted.RefreshingFlask, ProductsAccepted.Broadsword to Isolated Traveller to indicate that the items that the Isolated Traveller can purchase.
    a. Pros
        i. The use of an enum (ProductsAccepted) to represent purchasable items follows the SRP principle by clearly defining and encapsulating the responsibility of item that can be sold to Isolated Traveller within the enum. Each enum value represents a single responsibility: indicating a specific type of purchasable item.
        ii. Multiple new Trader classes (such as various trader that can buy items from players) may have similar items that they can buy in the game. These classes can implement or inherit shared behaviours using capabilities without adding extra classes or duplicating code. This allows the prevention of using multiple if…else statements to check for an instance of a class as we can establish each of these classes with common capabilities. This follows the Open-Closed Principle, which dictates that classes should be open for extension but closed for modification.
        iii. This allows for future extensions as the same type of implementation can be used for a new Trader (Not an Isolated Traveller), as the new Trader just needs to have the capability Status.MERCHANT and ProductsAccepted.*ItemToBuy* which allows the new Trader to be able to buy these products from the Player

    b. Cons
        i. As the number of items that can be bought by the IsolatedTraveller increase, the number of capabilities required rises, which makes managing several capabilities for an actor can become challenging.
            - However, this trade-off is preferable to using an alternative implementation of instanceof type checks, which can lead to tight coupling between classes and make the code harder to maintain

3. For the implementation of the Player selling an item to an Isolated Traveller (PlayerSell), to determine the SellAction that the item can do on the

IsolatedTraveller/Trader, we implement a new interface ‘PlayerSell, to the items that can be sold to the IsolatedTraveller.

a. Pros
    i. Having a separate interface for ‘PlayerSell’ and ‘ and ‘PlayerPurchase’ allows us to abide by the Interface Segregation Principle as we do not have one God interface for each item which must implement a method even when the method has no relation or use to the item. Having separate interface allows the items to be independent on what it sells and purchases.
    ii. The Item can determine the price of the item to be sold to the Isolated Traveller and also determines if the item will be scammed by the Isolated Traveller or not while returning the Sell Action that the Item can do to Isolated Traveller when it is in one of its exits. Since the standard selling price of each item is common (constant) for every item regardless of the Trader and only only if certain probability condition is met the price changes, the implementation of having the each item access to the standard selling price of items as constants and every item that can be sold having the method ‘playerSellItem()’ to identify the finalPrice of the item and the state of the item (scammed or not) respects the Single Responsibility Principle. This implementation follows the SRP principle as it allows the Item to have full control on the actions that the item allows its owner to do to other actors and the Isolated Traveller to manage the standard price of each item.

b. Cons
    i. If similar selling actions are needed for different products, there might be some code duplication. Each product type might have its own implementation of the PlayerSell interface, leading to redundancy.
        - However, since almost all of the ‘PlayerSell have implementations specific to the type of item and in the context of future extensibility and code modifiability in the future the current implementation is a better trade-off to other alternative approaches.

Requirement 4 (Jun Hirano)

GreatKnife

1. The GreatKnife is a special weapon in the game. Players can use it to attack with the StabAndStep skill and then move to a safe spot. This weapon isn't just for fighting. Players can also buy or sell it to a merchant named IsolatedTraveller. It has a feature where its selling price might change because of a scam chance. The class knows its owner by using a special function called a tick(), and it follows rules set by the SellGreatKnife interface for selling actions.

   Based on Requirement 4, we developed the Giant Hammer. Its implementation is very similar to that of the GreatKnife. The sole distinction is that players cannot purchase the GiantHammer from merchants. Thus, in this design rationale, we'll primarily focus on the implementation of the GreatKnife and its associated skills.
   a. Pros
      i. The GreatKnife class adheres to the PlayerSell interface, ensuring that the weapon possesses the ability to be sold. By implementing this interface, the class is obliged to provide a concrete implementation of the PlayerSell(int sellingPrice) method, which serves as an abstract approach to the selling mechanism. This approach is in line with the Dependency Inversion principle because all Item access to the concrete sell method via interface rather than directory access Sell. The intricacies of the selling action are fully encapsulated within the separate Sell class. This separation enhances code readability, as the GreatKnife class remains uncluttered with the detailed logistics of the selling process. Moreover, the implementation of PlayerSell indicates it is designed with the Open/Closed principle in mind. Users can extend the selling behavior by implementing the interface for other items without needing to modify the existing codebase.

      ii. The StabAndStep class represents a sophisticated combat action, more intricate than the conventional AttackAction. To streamline its design, we have encapsulated the AttackAction and SafeStepBehaviour within it. While the AttackAction is responsible for damaging the enemy, the SafeStepMovement facilitates player movement after the attack. This encapsulation not only simplifies the design of StabAndStep but also enhances its readability and reliability.
   b. Con
      Currently, the decision to determine if it's a scam is being made within each item's sellToIsolatedTraveller() method. This approach might be suitable when there's only one type of merchant. However, if we need to introduce various types of merchants in the future, each with their own scam rates, this design is not suitable. To make the system more flexible, each merchant should possess its own scam rate information, rather than the item calculating it.

Giant Hammer, GreatSlamAction

1. The GreatSlamAction class encapsulates the special ability of an actor to perform an enhanced attack action in a game. When an actor uses a weapon to deal a "Great Slam" attack to a primary target, while also potentially inflicting damage on actors surrounding the primary target. The primary actor receives full damage from the weapon, and the surrounding actors receive half of that damage. This action will only be executed if the actor has sufficient stamina, and if executed, will consume a portion of the actor's stamina.
    a. Pro
        i. The GreatSlamAction class introduces a rich layer to gameplay, providing players with an additional strategic combat mechanism. By adhering to the Single Responsibility Principle, the class ensures that each method and operation within the class has a distinct and well-defined role, optimizing code maintainability and readability. This commitment to clean code design ensures that future modifications and expansions of the game's combat mechanics can be executed efficiently.
    b. Cons
        i. From the perspective of OOP and logic, the GreatSlamAction class showcases several potential areas of concern. Firstly, there's an evident violation of the encapsulation principle, as member variables like weapon, target, and direction are package-private, risking unintended modifications from outside classes within the same package.
        ii. Moreover, the method is lengthy and does multiple tasks, such as checking stamina, calculating damage, and handling multiple attack scenarios, indicating a potential breach of the Single Responsibility Principle

Requirement 5 (Everyone)

1. For Requirement 5, a WeatherControl class and a Weather Enum class were introduced. The WeatherControl instance is created within ForestWatcher. The Weather Enum class comprises the enums SUNNY, RAINY, and NORMAL. During each turn (ForestWatcher playTurn()), the switchWeather() method from WeatherControl is invoked, determining whether the weather should change based on the switchCounter value passed during instantiation. If yes, it updates the Weather.CURR enum and adjusts the capabilities of the ForestWatcher accordingly.

    a. Pros
        i. WeatherControl encapsulates the responsibility of managing and switching weather conditions, adhering to the Single Responsibility Principle. Having a class that controls the weather changes based on the counter setting rather than having it in the ForestWatcher class allows us to avoid having a God class which handles all measures. In addition, this is also extensible, as a new Boss Enemy who would want to control the Weather, is able to set the counter at which it wants to change the weather by setting the switch counter.
        ii. By defining weather-related behaviors using capabilities (e.g., Weather.SUNNY, Weather.RAINY), the same behavior can be applied to multiple actors or grounds without duplicating code.
        iii. This promotes the reuse of weather-specific actions across different entities in the game. This allows us to prevent Open-Closed Principle as we do not have to keep checking what is the instance of the Enemy, instead we can just specify that the Enemy has the Weather capability and the code will be able to identify that specific enemy.
        iv. ForestWatcher is also adhering to Open-Closed Principle as it extends Enemy and implements Follow

    b. Cons
        i. When we have multiple more weather, the code might get more complicated as we have to specify different conditions for different weathers
            - However, this trade-off is preferable to using an alternative implementation of instanceof type checks, which can lead to tight coupling between classes and make the code harder to maintain

2. WeatherControl can be instantiated in 2 ways, one way by only passing the parameter gameMaps which was used in Application to set the current weather (WEATHER_CURR) for all maps given in the gameMaps ArrayList. The second way to instantiate WeatherControl is by passing in an gameMaps ArrayList as before and also the actor which “controls” the weather and the int switchCounter (determines the turn number in which the weather would change).
    a. Pros

        i. Having 2 different constructors for common implementation of the code allows us to abide by the DRY principle. Although the first constructor does not have the ability to switch the weathers, it must be able to set the current weather to the gameMaps specified as opposed to the second constructor which can control the switching of weather. But since, the setting of the weather in the gameMaps utilise the same implementation of setting the weather to Sunny or Normal or Rainy, having such constructor implementation allows reusability adn code redundancy
    b. Cons
        i. There might be some methods that the 2nd constructor can control that the first constructor cant which mightresult in syntax error
            - To solve this problem, we have implemented a new variable, controller which does not allow the codes in switchWeather as the controller is set to False. This allows code reusability and avoids conflicts

3. To make the weather affect the grounds and actors in the boss room and Ancient Woods. ForestWatcher takes a parameter called gameMaps which is an ArrayList of GameMaps passed upon instantiation of ForestWatcher which is then passed into WeatherControl upon instantiation. In the playTurn of ForestWatcher, a check occurs if the ForestWatcher has the Weather.SUNNY or Weather.RAINY capability, depending on the result of the check it calls isSunny() or isRainy() from WeatherControl, these methods assign Weather.CURR the rightful weather enum and calls the method updateWeatherForAllMaps() while passing in the enum Weather.SUNNY or Weather.RAINY. This method then calls the method updateWeather() for each GameMap in the gameMaps ArrayList which removes the Weather.SUNNY and Weather.RAINY capability for each ground coordinate and actor in the GameMaps and adds the Weather enum that was passed in the method call.
    a. Pros
        i. The variable Weather.CURR is a static variable that is used to set the weathers for each spanning enemy. This is because every spawning enemy will not know what the current weather is. So adding the capability of Weather.CURR to the spawning enemy allows us to allow the enemies to spawn based on the right weather condition upon instantiation.
        ii. By assigning specific capabilities for each weather, the code avoids duplicating similar logic for different weather conditions (by not having the same repeated code for each weather of isSunny(), isRaiiny(), isNormal()). Each capability encapsulates unique behavior, ensuring that similar actions (e.g., updating the actor or ground status) are handled consistently across the codebase with teh use fo teh variable Wetaher.CURR which allows dynamic reusability of code. This abides by the DRY principle such that we dont repeat redundant codes
    b. Cons
        i. There's a possibility of tight coupling between components (e.g., ForestWatcher, WeatherControl) and the Weather Enum. Tight

coupling can hinder the flexibility and maintainability of the code, making future modifications more challenging.

- But since, the ForestWatcher heavily relies on the WeatherControl as that's the responsibility of the ForestWatcher having the implementation as such rather than having a God class (combination of WeatherCOntrol and ForestWatcher) is a better implementation

4. The changes for each affected actors and grounds for the Forest and Battle Map are as below:
    a. RedWolf damage change:
        i. Checks if the RedWolf has the Weather.RAINY or Weather.SUNNY capability if yes it updates the Damage multiplier
    b. EmptyHut and Bush spawning rate:
        i. Both EmptyHut and Bush extend SpawnableGround which contains a method called updateSpawnMultiplier which could be overridden in EmptyHut and Bush
        ii. updateSpawnMultiplier was overridden in EmptyHut and Bush which performed checks if the EmptyHut/Bush had the Weather.RAINY/Weather.SUNNY capability and updated the spawn multiplier accordingly
    c. Forest Keeper Healing:
        i. Checks every turn if it has the Weather.RAINY capability
        ii. If yes, the ForestKeeper's health is updated

    Having each Actor and Ground control each of their actions depending on the weather adheres to SRP where each actor or ground control their actions depending on their weather, Open-Closed Principle that prevents multiple if… else if these are controlled by ForestWatcher or WeatherControl and allows future extensions and modularity as the code can be modified easily