Design Rationale

By: MA_AppliedSession1_Group3

<u>REQ 5: Abxervyer, The Forest Watcher</u>

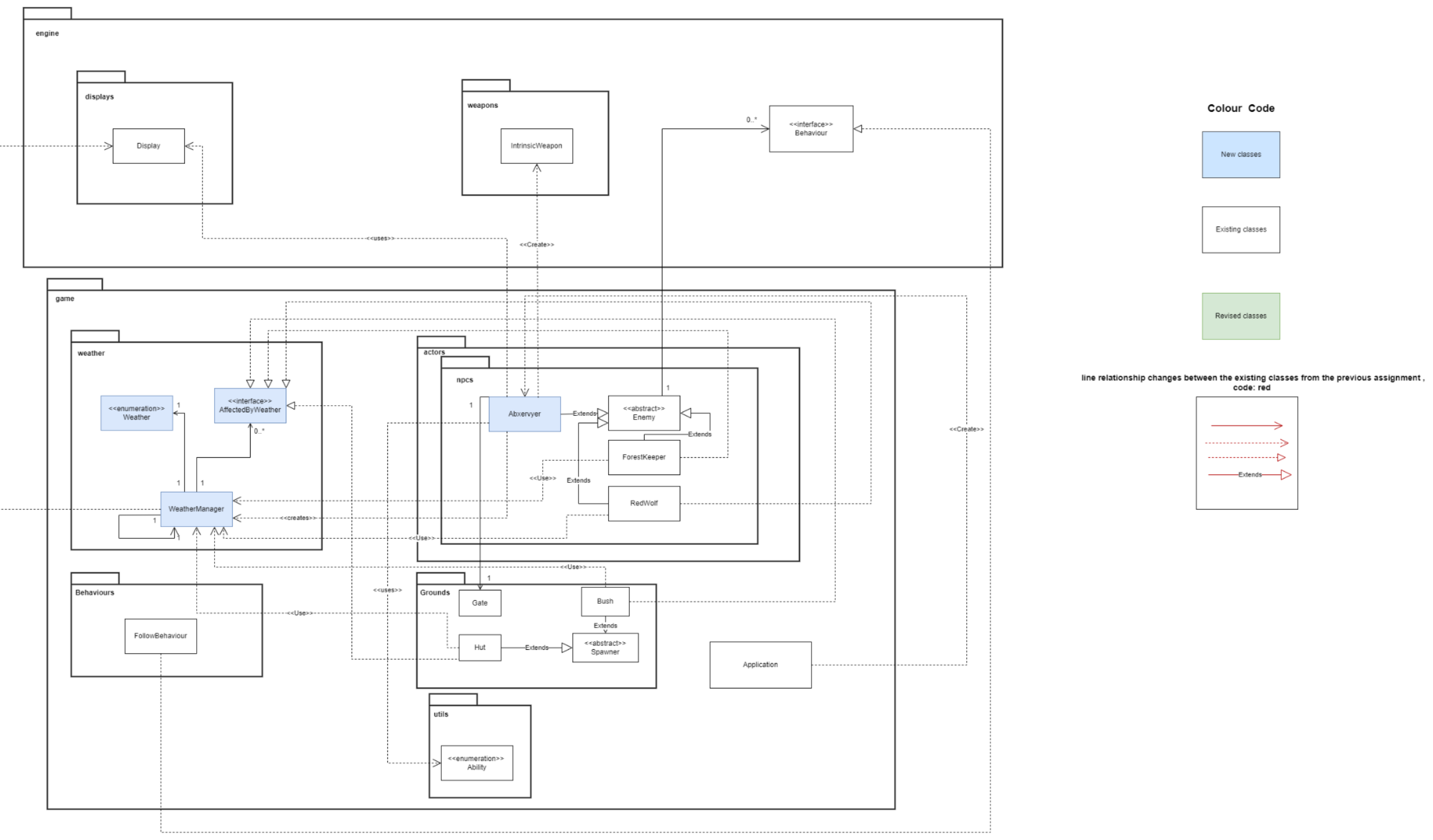

Figure 1: UML class diagram of REQ 5: Abxervyer, The Forest Watcher

The provided diagram shows the entire system implementation for Requirement 5, Abxevyer, The Forest Watcher. The primary goal of this design is to create a new boss enemy, Axervyer, the Forest Watcher. The boss can control weather in the forest, and some grounds such as hut and bush, as well as enemies such as Red Wolf and Forest Keeper will be affected by the weather. Thus, new classes: Abxervyer class,WeatherManager class, Weather class and AffectedByWeather interface class will be created and these classes will interact with the existing classes in the extended. system

Most of the implementation of the boss class is similar to the other enemies and does not require much explanation. It can also follow the player, so it is also necessary to add a FollowBehaviour to its 'behaviours' at construction time. according to the rules, when the boss is defeated, it drops Runes and its current location becomes a gate. The logic for both should be written in the dropItem method, adding Runes (5000) to the boss location, and setting the Ground at this location to Gate (which needs to be specified at boss creation time)

We can add a new static variable "BOSS_DEFEATED" to the FancyMessage class, so we can display this string in the overridden unconscious method of the boss class.

Before that, what all enemies do at the start of their turn can be generalized as "choose what can be done based on priority until an action has been performed." What about on the boss's turn, where things change slightly, as the boss will change the weather, so when the boss starts the turn, it will change the weather first, and then perform what an enemy would do normally. So, the boss needs to override the playTurn method in the Enemy class, and within the method, the first part is to change the weather, and the second part is to call Enemy's playTurn. The boss's turn is randomized and not necessarily right after the player (i.e. the boss isn't necessarily the first enemy that can start the turn), and the weather change is something done at the very beginning of the boss's turn, so it's possible to have both aggressive red wolves and non-aggressive red wolves, etc., in a single turn.

While we can change the spawn rate of Spawner (Ground type) using setSpawnPercentage, we can't go back to the normal rate. Therefore, we need to add a new attribute 'iniSpawnPercentage' to Spawner, and add the corresponding getter method to represent the original spawn rate, i.e., when we want to set the rate back to the original, we just need to setSpawnPercentage(getIniSpawnPercentage()).

The above are the points that our group discussed in a meeting prior to writing REQ5. Then comes the most important part, the implementation related to changing the weather.

REQ5 indicates that there are two weathers, Sunny and Rainy, and at first, we wanted to use a boolean value for the weather. With a boolean, we're limited to only two states (true or false). If we ever want to expand the types of weather (e.g., sunny, rainy, snowy), we'll need to change the data type and refactor the code. And the change will be huge. Also, a boolean value doesn't provide much information about the state itself. It doesn't convey the meaning or context. Thus, instead of using boolean values to toggle the weather, we decided to introduce an enum class to represent the type of weather. Enums allow us to define a set of named constants, which can represent different states or options. This makes it easy to extend and add more weather types in the future. Enums provide a clear and self-documenting way to represent states. We're effectively encapsulating that set of values in one place. This helps to keep the responsibility of managing those constants separate from other parts of our code. Therefore, it follows Single Responsibility Principle (SRP) and Open/Closed Principle (OCP).

We can't put all the logic for controlling the weather in the boss class, such as keeping track of the number of turns, changing the weather every three turns, and displaying the current weather every turn. By including weather control logic in the boss class, we're increasing its complexity and coupling it with an unrelated responsibility. This can make the code harder to understand and maintain. To address this, we could consider creating a separate WeatherManager class that is responsible for managing the weather and the boss can interact with this manager class. By separating concerns in this way, we adhere to the Single Responsibility Principle.

There is a method run in WeatherManager that affects entities like RedWolf, ForestKeeper, Hut, and Bush according to the current weather. Along this line of thought, we know that the manager needs to be able to access all of the entities that will be affected by the weather. Therefore, we need an interface AffectedByWeather, and all entities that implement this interface are considered to be individuals that are affected by the weather (this interface is equivalent to a marker). Let the WeatherManager class have an ArrayList attribute, a collection that holds all the entities that implement the affectedByWeather interface. In this way, the weather manager will be able to access all affectable entities.

How to add all the objects in the game that implement the AffectedByWeather to this ArrayList? We can have a method, registerWeather(), to add an AffectedByWeather "object" to the ArrayList. And this method can be called in RedWolf's constructor, ForestKeeper's constructor, Bush's constructor and Hut's constructor. This ensures that whenever a new object is introduced into the game, it will be added to the manager's ArrayList. In order to call register (), we need to create the WeatherManager object, and we need to make sure that we are creating the same object, we need to use the singleton pattern. This can be useful for centralizing weather management and ensuring that all parts of the game are interacting with the same WeatherManager. At this point, we realized that unregisterWeather() is also necessary. When an enemy affected by weather dies, it needs to be removed from the list. And this can be done in the overridden unconscious (actor, map) and unconscious(map) methods.

Let's back to the AffectedByWeather interface, it has two methods, affectedBySunny() and affectedByRainy(), which will be implemented by the Hut class, Bush class, RedWolf class and ForestKeeper class as those classes can This means that all four concrete classes need to override these two methods. Different types of entities can have their own unique reactions to changes in weather. This design allows for easy addition of new entities affected by weather without modifying the WeatherManager class directly. This design aligns with the Open/Closed Principle (OCP) and Interface Segregation Principle (ISP). Thus, our system can lead to more flexible and maintainable code.