FIT 2099 Assignment 1

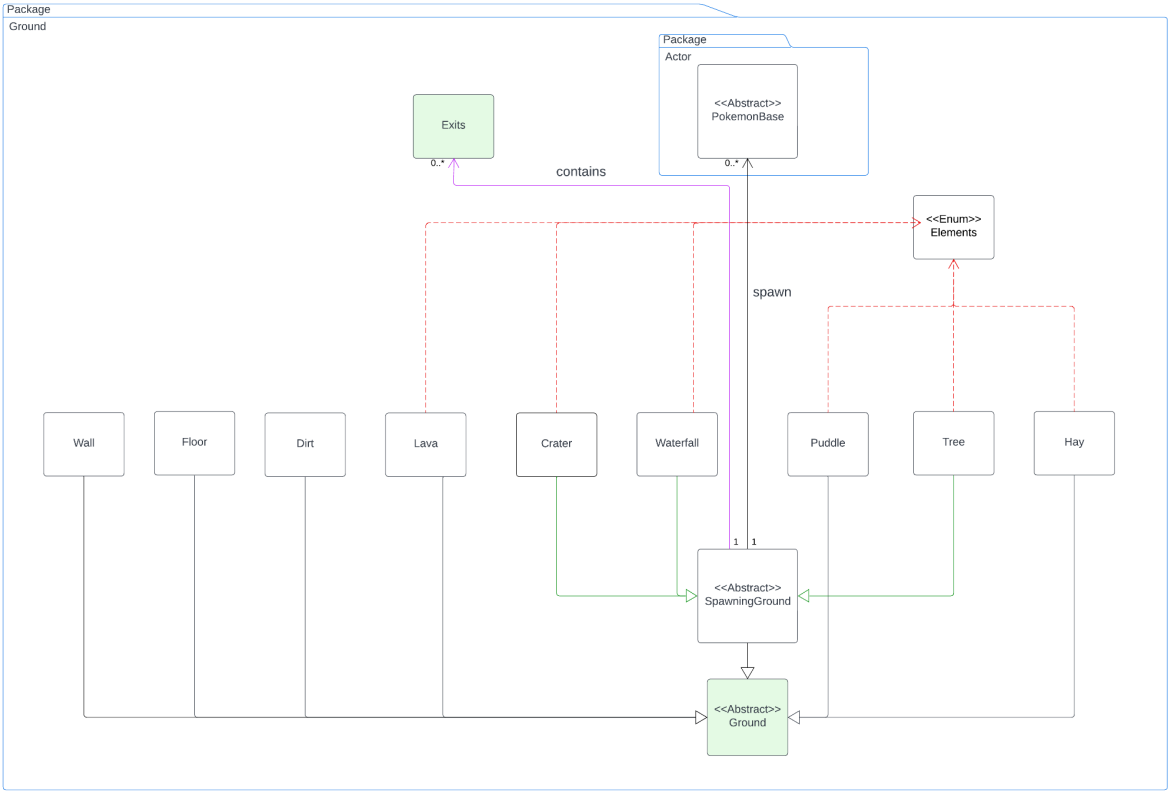
This assignment is entirely done by me (Zecan Liu). My group member Minh Anh Phan did not reply to our message since the assignment started. Another group member Ethan Kouris was assigned to do Req 2 and 4 but he already dropped this unit and informed me on 11/12/2022. He did not provide any expected work by deadline as well. Therefore, this document only covers Requirement 1,3 and 5 of the assignment.

Contribution Log Link:
FIT2099 - CL_Team17 - Assignments' Contribution Logs

Design rationale:

This project is a Rogue-like Pokemon game which consists of multiple packages.

REQ1 - Ground:



Design Goal:

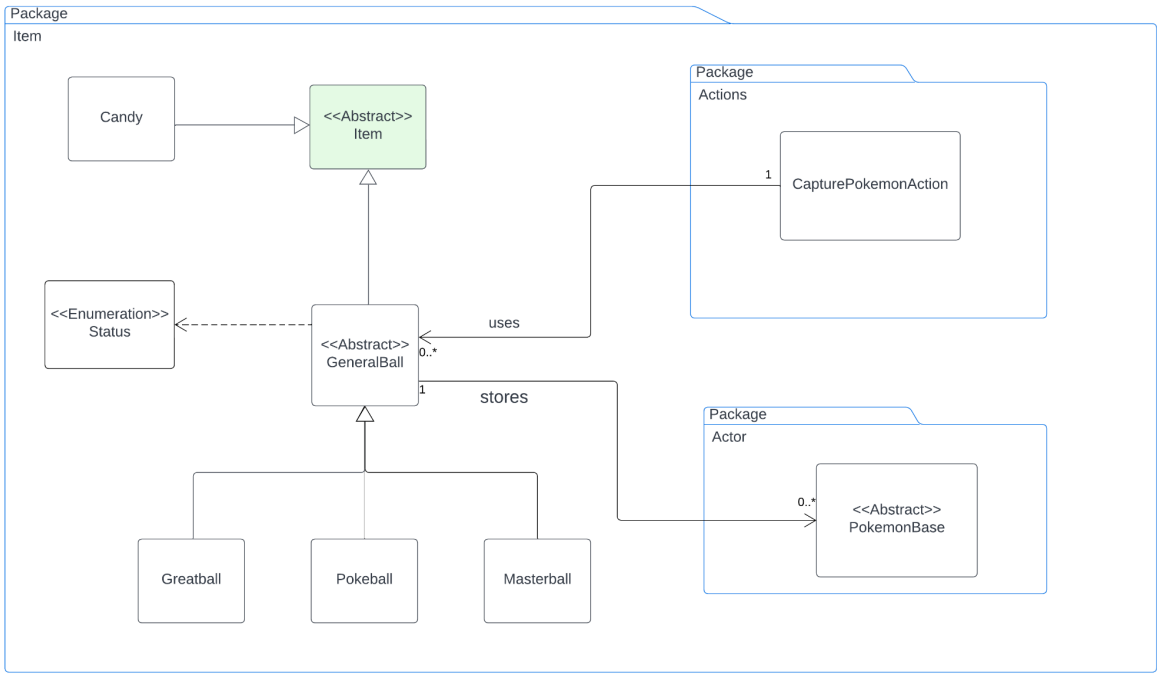
There are different types of grounds in this game. To achieve the extra functionality, I created new classes to extend the existing abstract **Ground** class.

To apply the **Single Responsibility Principle**, each of the different types of ground should be a class. Therefore, I created different classes for each type of ground.
To avoid repetitions, each of the ground types extended **Ground** class directly or indirectly because they share some common attributes and methods.
Wall, Floor and **Dirt** are the basic types which do not have elements or special abilities, for example, spawning a Pokemon. Therefore, they only extend the abstract **Ground** class. **Lava, Puddle** and **Hay** classes have elements, so not only they extend the abstract **Ground** class, but also have a dependency relationship with the enumeration class **Elements**. **Elements** class contains the 'Fire', 'Water' and 'Grass' elements. By using the addCapability method in Ground class, they can add different elements to their objects and store this attribute in capabilitySet.
Crater, Waterfall and **Tree** contain elements as well, and they have extra ability, which is to spawn a Pokemon under certain conditions. Therefore, I created another abstract class called **SpawningGround**. This abstraction has eliminated the repetition on the action of spawning Pokemons. This class extends the **Ground** class as it should have the same basic functionalities. **Crater, Waterfall** and **Tree** extend **SpawningGround** to implement the abstract method to spawn different Pokemons. Alternatively, I have considered making **SpawningGround** as an interface. However, there are extra conditions on spawning Pokemons for **Waterfall** and **Tree** classes. They need to get access to its surrounding locations and check if there are any same element grounds. So they should have a method called getSurrounding(), which returns a boolean value to check if the spawning conditions have been met or not. **Crater** class, on the other hand, does not have this requirement. Therefore, this alternative approach has been abandoned as the classes which implement an interface should implement all the methods included in the interface.
Another reason is that the action 'Spawn a pokemon' is turn based, which means that every time the tick method runs, it should determine whether or not to spawn a Pokemon. Therefore, having an abstract 'spawn pokemon' method inside the tick method of **SpawningGround** reduces the repetition. **Liskov Substitution Principle** has been applied as **Crater, Waterfall** and **Tree** classes do not change **SpawningGround** existing functionalities**.**

SpawningGround has an association relationship with **Exit** class as **Waterfall** and **Tree** classes need to loop through the surrounding locations to check if there are any same element grounds. Therefore, it has an attribute which is a list of Exits.

Instead of having the association relationship on **Crater, Waterfall** and **Tree** with **Torchic, Mudkips** and **Treekob**, the abstract **SpawningGround** class has an association relationship with the abstract **PokemonBase** class is more appropriate. **Dependency Inversion Principle** has been applied in this design as the dependency is on abstractions. This approach made extension more easily, it won't affect existing code if introducing new Pokemons or adding new types of grounds.

REQ3 - Items



Design Goal:

Since REQ 6 is not required in this circumstance, the main items of this game are the balls used to capture and store Pokemons. Classes are created based on different types of items.

To apply the **Single Responsibility Principle**, all three types of balls should be a concrete class as the effectiveness on capturing Pokemons differs for these three balls.

Since **Pokeball** & **Greatball** & **Masterball** classes share some similar behaviours, for example, capture and store Pokemons, it's reasonable to create an abstract class called **Generalball** and let the three classes extend it.To avoid repetition, the abstract **Generalball** class extends the abstract **Item** class as the methods in **Item** class will be used during the game.

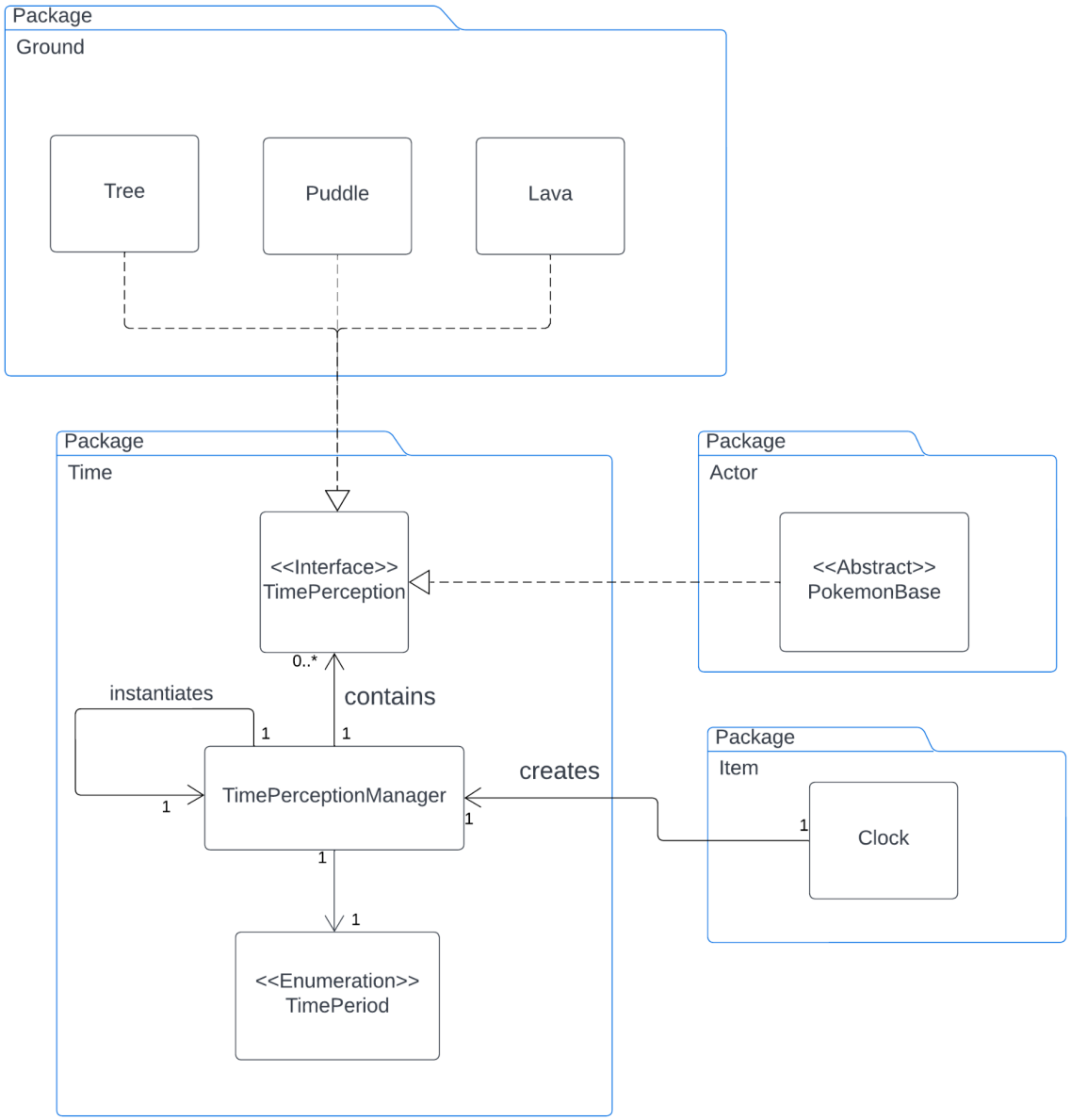
Alternatively, I have considered making the **Pokeball** class extend **Item** class and **Greatball** and **Masterball** extend the **Pokeball** class. However, this design violates **Liskov Substitution Principle** as we are unable to replace the **Pokeball** class with **Greatball** / **Masterball** classes because they have different affection points requirements when capturing Pokemons. Therefore, this alternative approach has been abandoned.

After the player uses Pokeballs to capture Pokemons successfully, the Pokemon will be stored inside the balls. Therefore, **GeneralBall** has an association relationship with the **PokemonBase** class. This approach follows the **Dependency Inversion Principle** as it does not have an association relationship with **Torchic, Mudkip** and **Treecko** directly. It won't affect existing code if introducing new Pokemons.

Since **Greatball** and **Masterball** can be used to capture Pokemons more effectively, this difference can be set by adding new elements in **Status**. To be more specific, **Status** contains extra three elements and can be set as an attribute on those three classes respectively. So when a player decides to capture a Pokemon, a method should check the affection points and decide which ball is able to use by selecting from the Enum class. These elements can be added to the ball classes respectively by using the addCapability method. Doing such, when a player acquires a Pokeball/Masterball/Greatball and stores it in its inventory, this capability will be automatically added to the player, due to the reason that **Player** extends **Actor** and **Actor** implements **Capable** interface as well.
Candy class directly extends the abstract **Item** class as it has different behaviour compared to Pokeballs.

The **CapturePokemonAction** class will use Pokeballs to capture Pokemons, therefore, it has an association relationship with **GeneralBall**. This follows the **Open Closed Principle** as if adding new types of Pokeballs which extend **GeneralBall,** it won't affect existing functionality.

REQ5 - Time



Design Goal:

The game is turn-based and some of the ground types & Pokemons need to experience the time effects. Therefore, classes are created in order to implement this functionality.

To apply the **Single Responsibility Principle**, I created a The **TimePerceptionManager** is a singleton class where only one object exists in the game. It is used to control the day and night effect applied to different objects. I created a **Clock** class which used to override the tick method and have a counter attribute to keep track of the game turns.

The counter will be used by **TimePerceptionManager** to determine whether it is day or night. **TimePerceptionManager** instance will be created inside the Clock class. Alternatively, I have

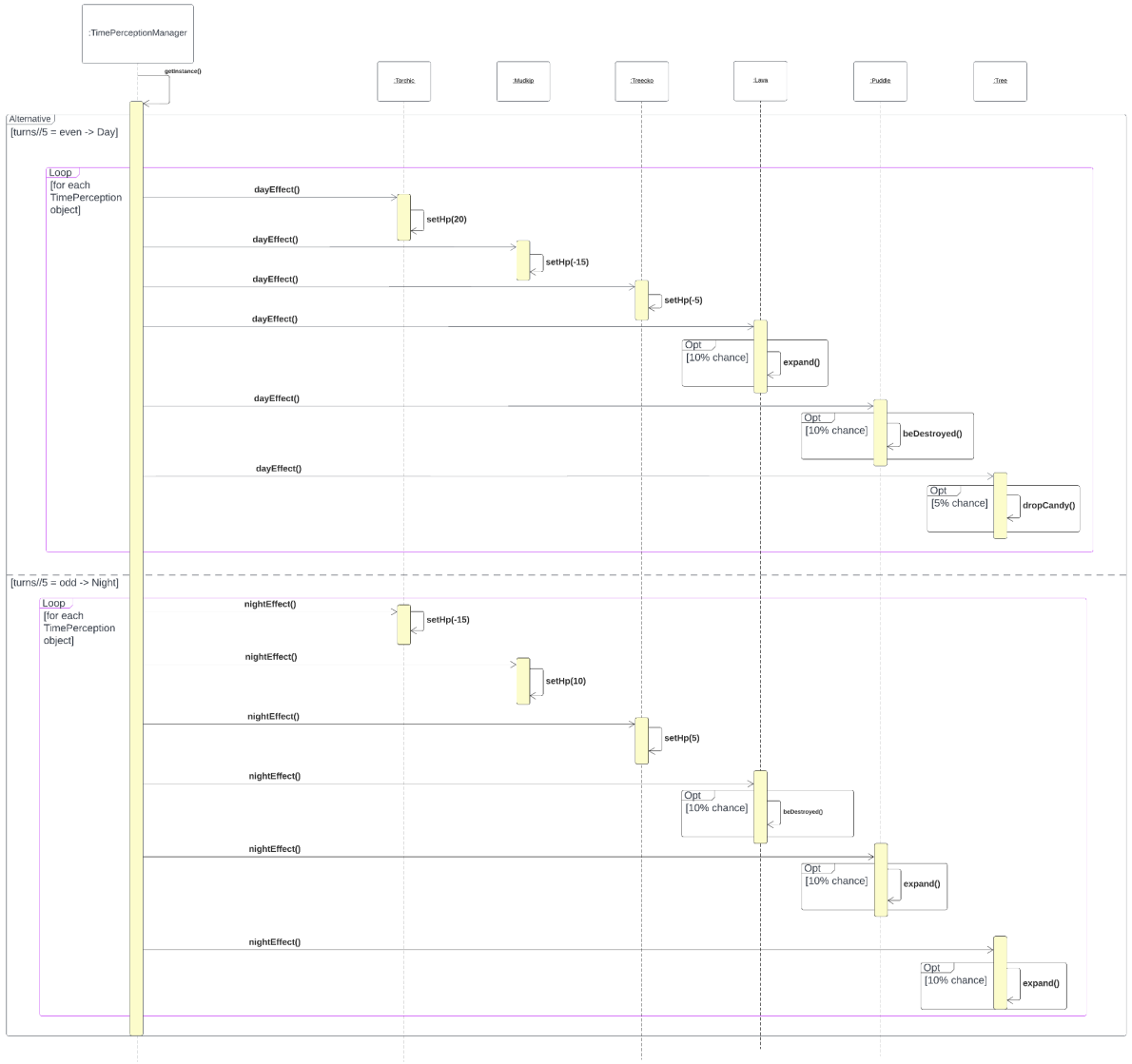
considered letting the existing **Ground**/**Item** subclass be associated with **TimePerceptionManager** and place a counter inside the tick method to keep track of the turns. However, it will violate the **Single Responsibility Principle** as it will have multiple responsibilities, ie. control the time of the game. So this approach has been abandoned.

TimePerceptionManager contains objects which can be looped through to experience time. In this case, pokemons and three types of grounds have day and night effects. Instead of making **TimePerceptionManager** having several lists for objects of each class, I created an interface **TimePerception** and let those classes implement this interface. This design applied the **Dependency Inversion Principle**, as doing so **TimePerceptionManager** only contains one list that stores all the class objects which implement the **TimePerception** interface. The advantage of this interface is that if any other class also needs to apply day/night effect, the only thing it needs to do is implement the interface, and it won't affect the **TimePerceptionManager** (no need to modify existing code)**.** This design also applied the Open Closed Principle, as **TimePerception** is not a concrete class. If it's a concrete and parent class of the other classes, **TimePerceptionManager** needs to use a lot of 'if' to distinguish its child classes. It will be very difficult to maintain if adding new subclasses.

Enum class **TimePeriod** contains two types of the turns, which are day and night. It is associated with **TimePerceptionManager** as it will be used to classify based on turns.

Sequence Diagram

Below sequence diagram is created based on the **run()** method inside **TimePerceiptionManager** class.



Below sequence diagram is created based on the **tick()** method inside the **Waterfall** class. Each turn the tick method will be run for once, therefore, it shows how the **Waterfall** spawns a **Mudkip** under certain conditions for each turns.