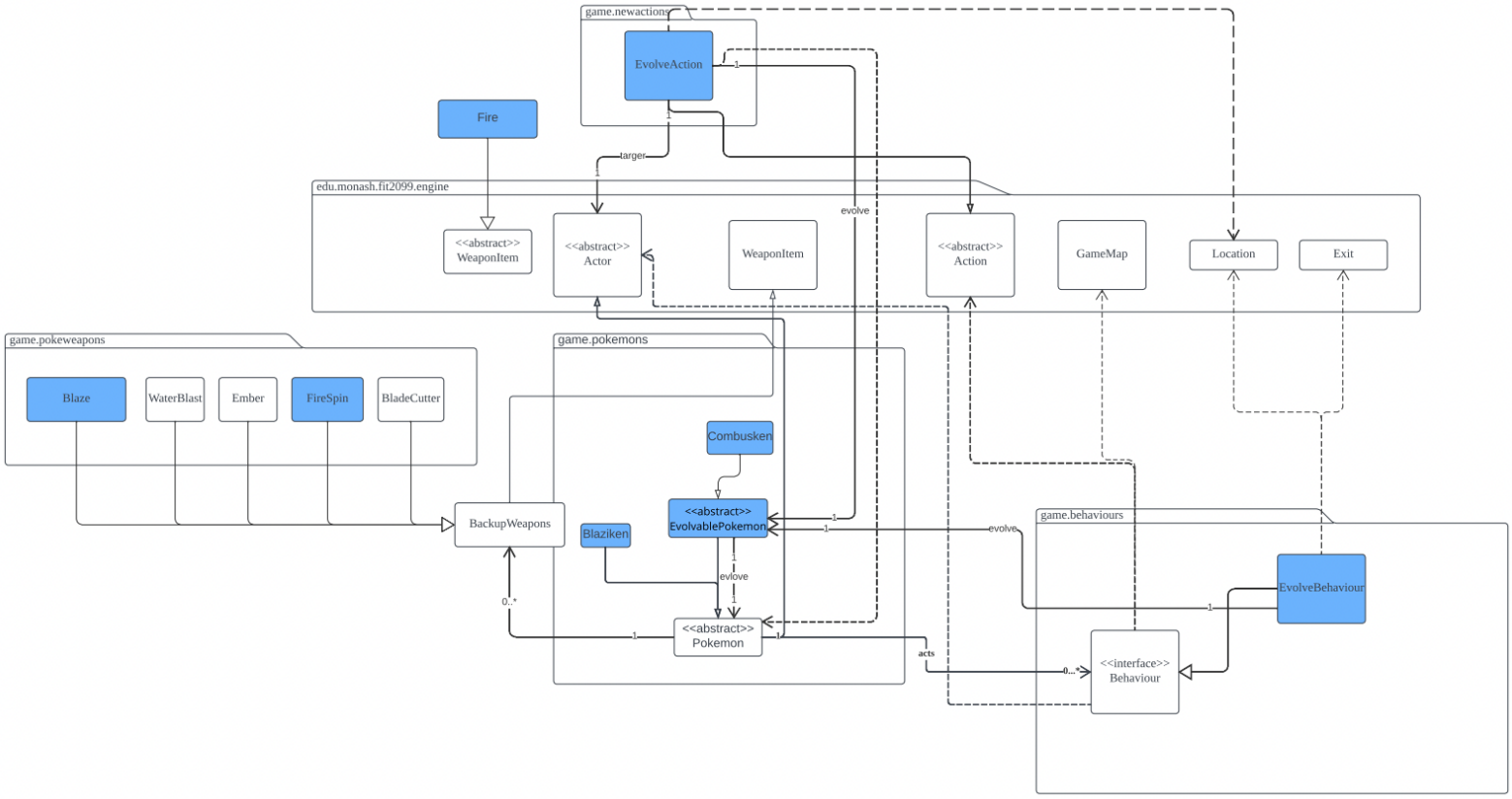
Assignment 3: Paul-Antoine Galliot & Zirui Liu

REQ1 Design Rationale:



New classes to be added:

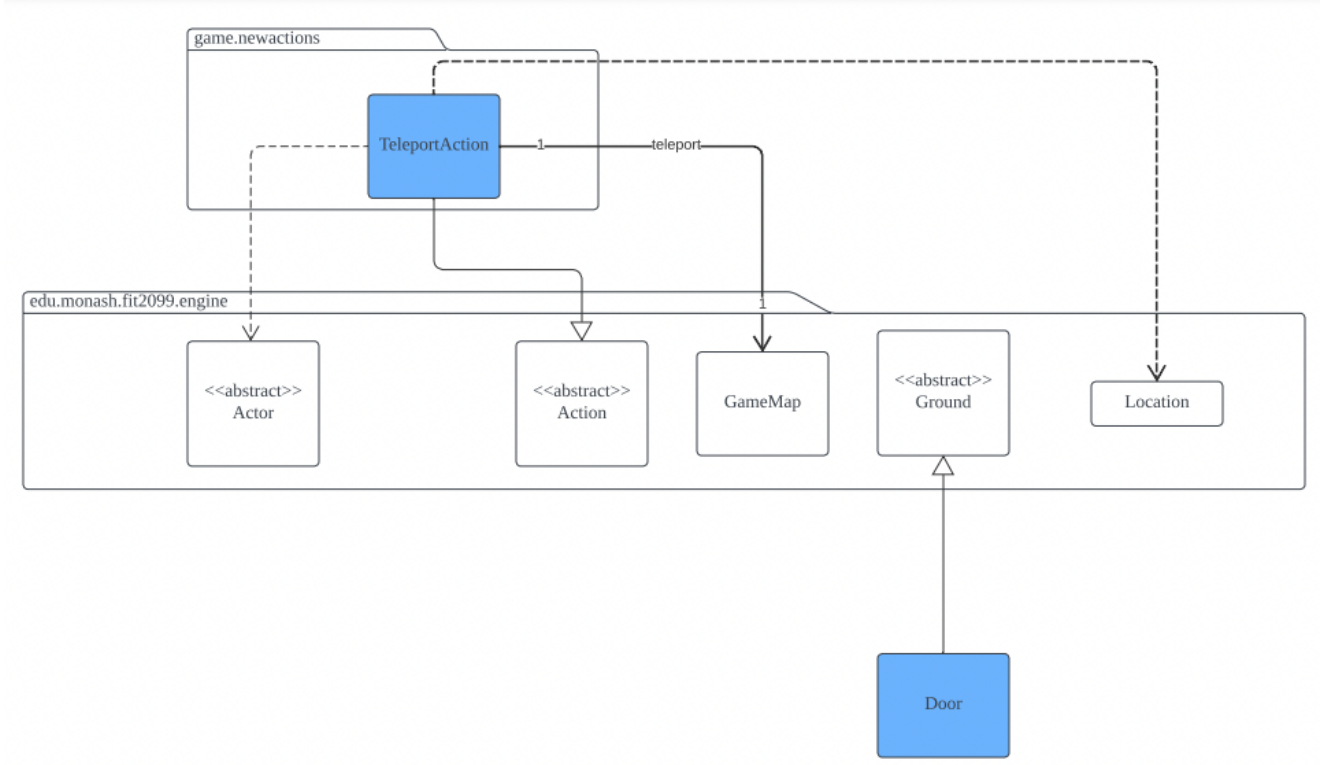
- EvolvablePokemon<<abstract>>
- Combusken
- Blaziken
- EvolveAction
- EvolveBehaviour
- FireSpin
- Blaze
- Fire

Reasoning REQ1:

For requirement one where I had to implement the evolution functionality I decided to add a new abstract class “EvolvablePokemon” which would handle the differences between a regular pokemon and one that can evolve. All pokemons extend this class except Blaziken as he is the only pokemon at the peak of his evolution hierarchy thus he only extends the “Pokemon” class. The purpose of adding a new abstraction is to adhere to the Single Responsibility Principle as the “Pokemon” and “EvolvablePokemon” each have different purposes thus if we only had the “Pokemon” class it would violate this principle. To accomplish this functionality I added a new action called “EvolveAction” that would handle the process of evolution and a behaviour

“EvolveBehaviour”. These two additions permit the evolution done by the actor and the pokemons. The last class that was added in this requirement was the class “Fire” which extends the abstract class “WeaponItem” and is used when Blaziken equips the “Fire Spin” special attack. To not violate Liskov Substitution Principle I added a status called “FIRETICK” which permits me to check if an item should be dealing damage to the pokemon instead of using instanceOf. Two other classes were also added which are “FireSpin” and “Blaze” which are the special attacks of the two new pokemons.

REQ2 Design Rationale:



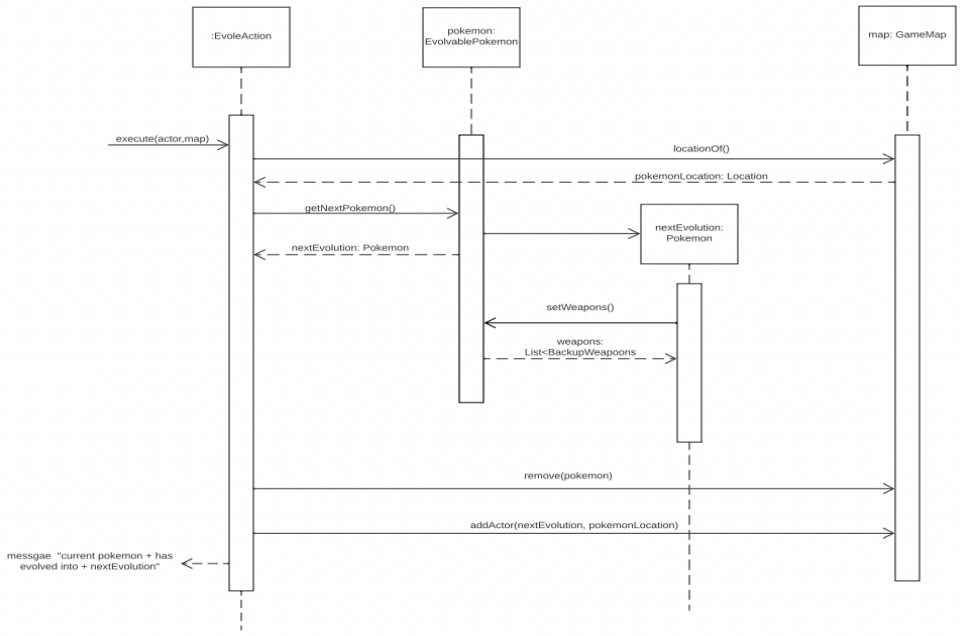
New classes to be added:

- TeleportAction
- Door

Reasoning REQ2:

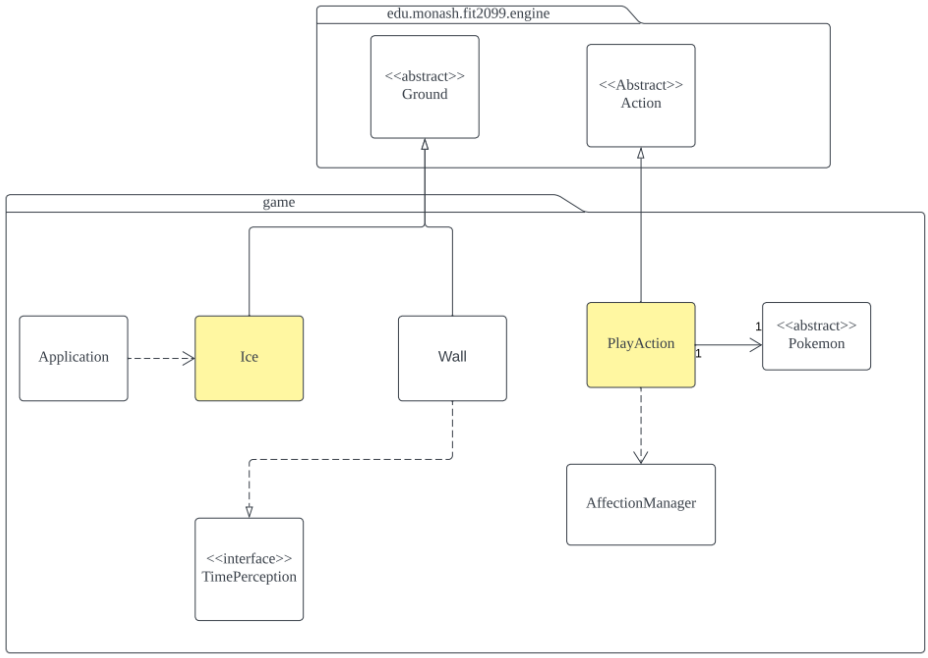
This requirement required two new classes which were “TeleportAction” which determines the logic that is needed to be used in order to complete the action. The other is the class that was needed to implement the ground “Door” which is the only ground in the game that has the capability of being able to teleport. To identify the door ground is used a status “TELEPORTABLE” which permits the actor to know if they can teleport from this particular ground. By using this I avoided violating Liskov’s Substitution Principle. This status is also helpful to determine where the actor will land in the next map otherwise those would have to be manually set which would not work well with any change to the map.

Interaction Diagram: EvolutionAction



Interaction Diagram Rationale:
This interaction diagram illustrates the logic behind the evolution action which simply retrieves the next evolution object, passes the weapons from the previous pokemon to the evolution through the setWeapons weapons. This method works by iterating through the possible weapons and appending them to the next evolution. The execute method finishes by removing the base pokemon from the gamMap and adding the evolution where the previous one stood.

UML Diagram REQ3



FLY:

1. Fly means the wall can be entered by the Player at a specific time. So the logic of checking if an actor can enter the wall is implemented in the canActorEnter() method in the Wall class **instead of** implementing it in the Player class. This approach works better as it adheres to **single responsibility principle** as the enter-enabling feature is the responsibility of the Wall class but not the Player class.

Instead of using the existing Status IMMUNE to identify the Player, we added a capability called FLYABLE to the Player. The wall will check if the actor has this capability using if statements. If the actor is FLYABLE and the time is fly time, then allow the actor to enter the wall.

This approach works better as it can avoid some issues if we will have an immune but flyable actor in the future. This approach also adheres to the single responsibility principle as the Player only has the responsibility of adding Flyable capability rather than enter-checking.

2. **Instead of** creating a new time period and making the Player can fly in the new time period, we choose to use the existing day and night effect to ensure the player can fly in certain turns. The dayEffect in Wall is used to disable the fly-in feature and nightEffect is used to enable it (i.e. remove/add capability of ENABLEFLY Status).

PROS:

This approach works better as it adheres to **Liskov Substitution Principle** as all classes/methods implementing the TimePerception interface have functionalities related to day and night time perception. Every class that implements dayEffect must have some functionality inside a nightEffect simultaneously. Also, all these functionalities are related to day/night time perception. As the Wall class implements TimePerception, we should be able to replace TimePercerption with Wall(and all other Classes that implement TimePerception) without disrupting the behaviour of our program. This also indicates this feature adheres to **Liskov substitution principle**.

CONs: No cons currently.

ICE:

3. **Instead of** using allowableActions() method in the ice ground, I implemented the logic of slipping feature in the tick() method of the Ice class. The logic includes checking if the actor has SLIP capability and using a loop to find an exit that the actor can enter.

PROs:

This approach works better as the tick() method is called every turn automatically. This approach makes slipping happen automatically(simple and easy) rather than using the allowable action in a complex and manual way. This approach adheres to the KISS principle as it makes the code simple and easy.