FIT2099 Assignment 1: Design

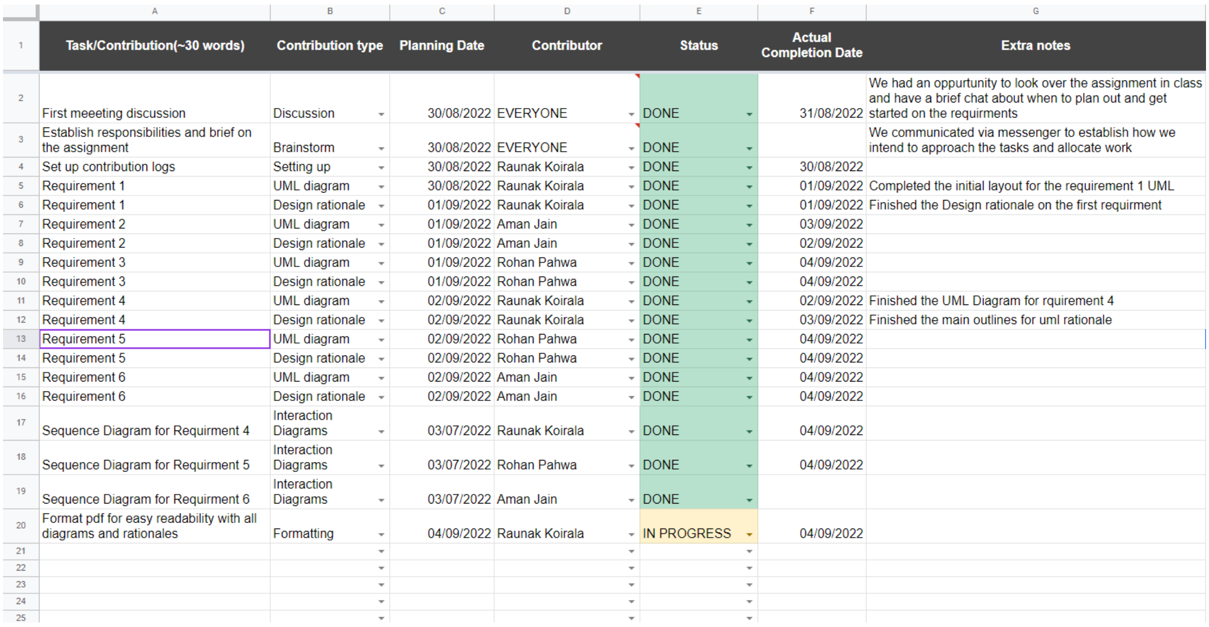
Group members:

Raunak Koirala

Rohan Pahwa

Aman Jain

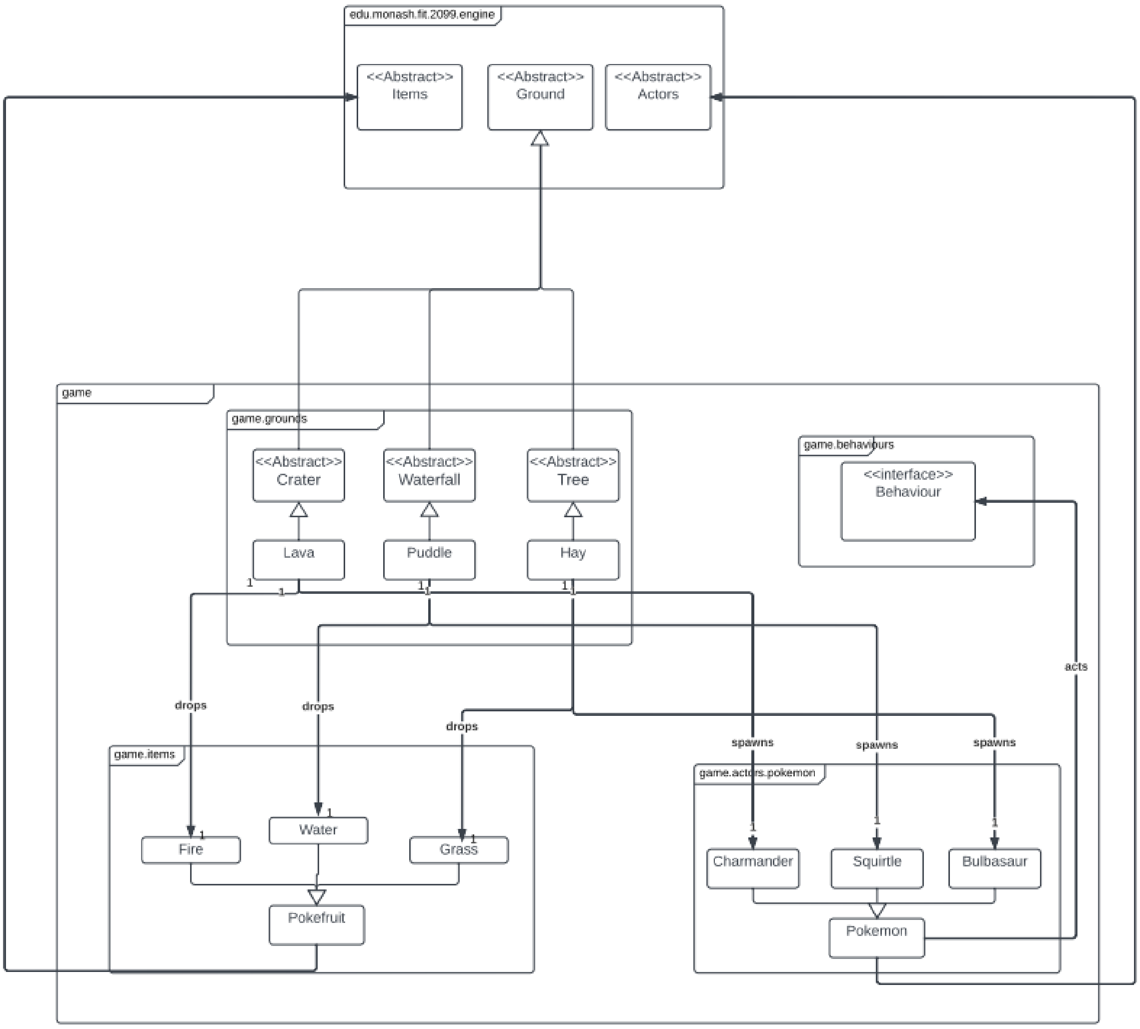
https://docs.google.com/spreadsheets/d/1z3_UeP_lR0X4kZdN-PbN7SxJbu9lY6RN7ykJtKTF9Rw/edit?usp=sharing



Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes
First meeeting discussion	Discussion	30/08/2022	EVERYONE	DONE	31/08/2022	We had an oppurtunity to look over the assignment in class and have a brief chat about when to plan out and get started on the requirments
Establish responsibilities and brief on the assignment	Brainstorm	30/08/2022	EVERYONE	DONE		We communicated via messenger to establish how we intend to approach the tasks and allocate work
Set up contribution logs	Setting up	30/08/2022	Raunak Koirala	DONE	30/08/2022	
Requirement 1	UML diagram	30/08/2022	Raunak Koirala	DONE	01/09/2022	Completed the initial layout for the requirement 1 UML
Requirement 1	Design rationale	01/09/2022	Raunak Koirala	DONE	01/09/2022	Finished the Design rationale on the first requirment
Requirement 2	UML diagram	01/09/2022	Aman Jain	DONE	03/09/2022	
Requirement 2	Design rationale	01/09/2022	Aman Jain	DONE	02/09/2022	
Requirement 3	UML diagram	01/09/2022	Rohan Pahwa	DONE	04/09/2022	
Requirement 3	Design rationale	01/09/2022	Rohan Pahwa	DONE	04/09/2022	
Requirement 4	UML diagram	02/09/2022	Raunak Koirala	DONE	02/09/2022	Finished the UML Diagram for rquirement 4
Requirement 4	Design rationale	02/09/2022	Raunak Koirala	DONE	03/09/2022	Finished the main outlines for uml rationale
Requirement 5	UML diagram	02/09/2022	Rohan Pahwa	DONE	04/09/2022	
Requirement 5	Design rationale	02/09/2022	Rohan Pahwa	DONE	04/09/2022	
Requirement 6	UML diagram	02/09/2022	Aman Jain	DONE	04/09/2022	
Requirement 6	Design rationale	02/09/2022	Aman Jain	DONE	04/09/2022	
Sequence Diagram for Requirment 4	Interaction Diagrams	03/07/2022	Raunak Koirala	DONE	04/09/2022	
Sequence Diagram for Requirment 5	Interaction Diagrams	03/07/2022	Rohan Pahwa	DONE	04/09/2022	
Sequence Diagram for Requirment 6	Interaction Diagrams	03/07/2022	Aman Jain	DONE		
Format pdf for easy readability with all diagrams and rationales	Formatting	04/09/2022	Raunak Koirala	IN PROGRESS	04/09/2022	

Class Diagrams & Sequence Diagrams - Requirements:

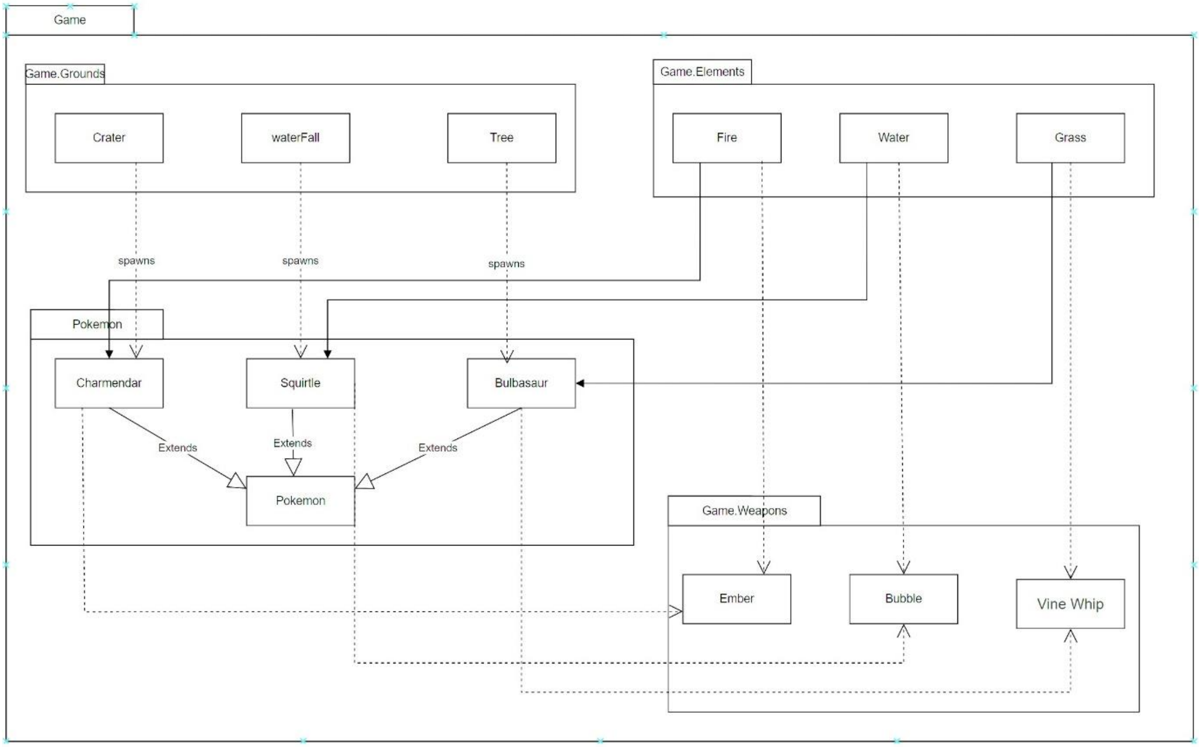
Requirement 1:



UML Requirement 1 Rationale:

For requirement 1, I decided to make the Crater, Waterfall, and Tree classes all abstract and extend them to Lava, Puddle and Hay classes respectfully in order to make use of the inheritance concept. I also decided to add a class Pokémon in order to have a parent class for all Pokémon as they will all have similar functions. Through using these abstract classes, instead of writing similar code for each class, we can use the code within the abstract class and utilise those methods in the others. This ensures that code isn't repeated and is easier to maintain. The crater, waterfall and tree classes are similarly extended from the ground abstract class as they all have similar functionalities and although they share the same parent class, they still maintain different characteristics thus adhering to the Single Responsibility Principle (SRP) and the use of extending from abstract classes also ensures that the Open-Closed principle isn't violated by improving maintainability and extensivity.

Requirement 2:



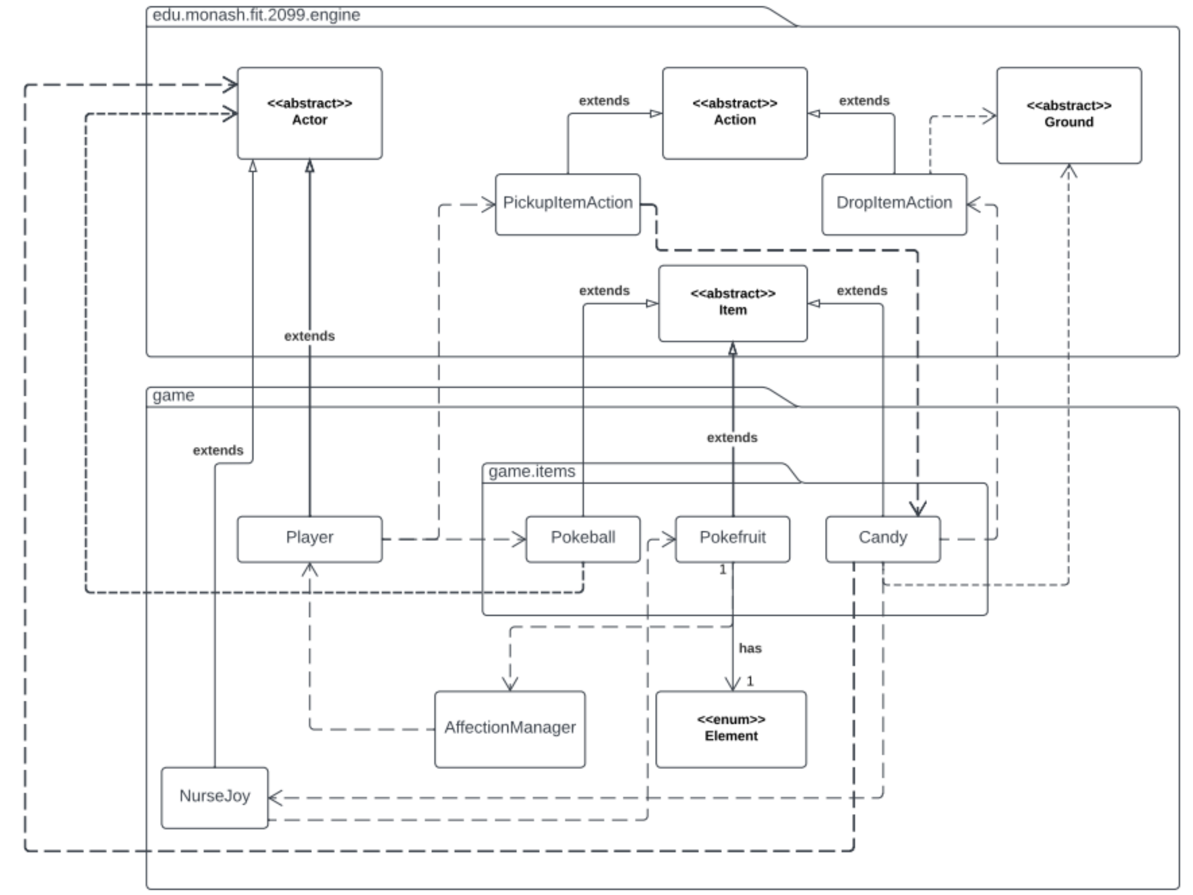
UML Requirement 2 Rationale:

For this requirement, the classes: Charmander, Squirtle, and Bulbasaur inherits all the properties of the Pokémon class. According to the requirement, brief Charmander spawns from Crater ground hence there is a dependency relationship between them, same goes with the waterfall and Squirtle classes and also the Tree and Bulbasaur classes.

The class fire is associated with Charmander because he is fire-type Pokémon and so is Water and Squirtle and Grass and Bulbasaur.

The weapons are dependent on the Pokémon types and the ground they are standing on, as described in the assignment brief. For example, the weapon Ember can only be equipped by Charmander given the condition he is standing on its type of ground i.e., Fire. The same is the case with Bubble and Vine Whip.

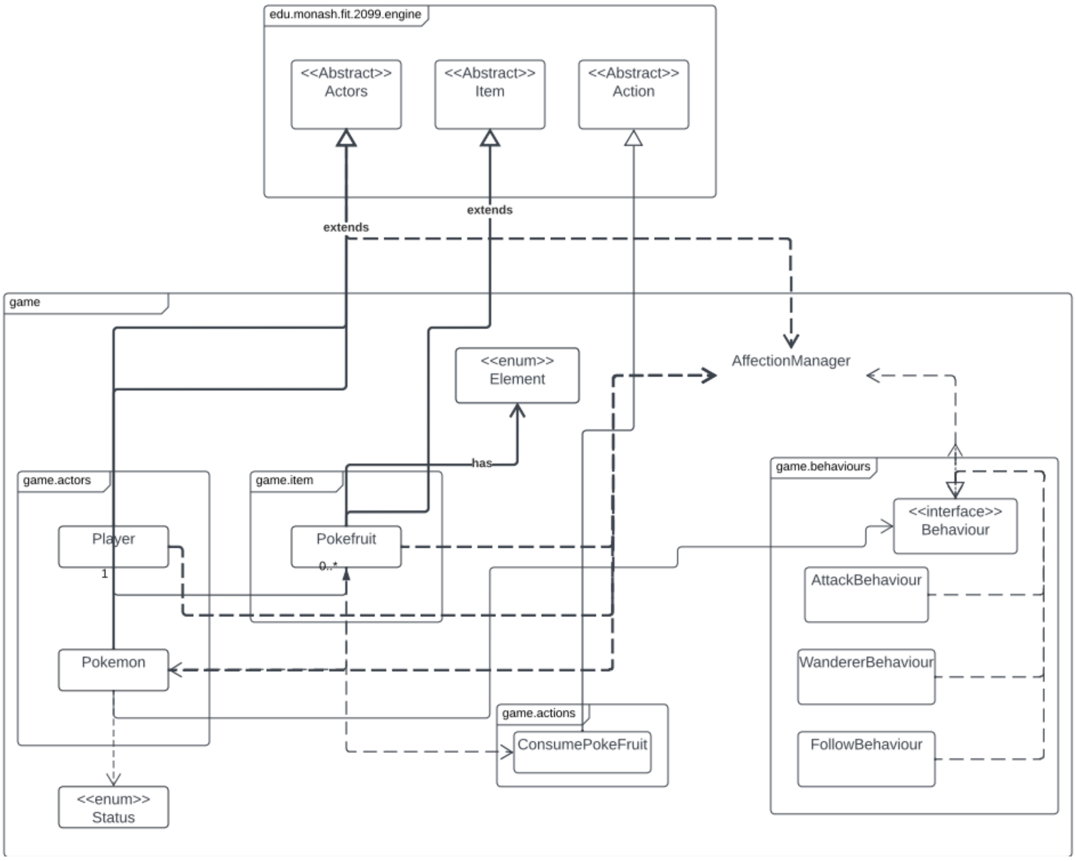
Requirement 3:



UML Requirement 3 Rationale:

The Requirement 3 UML features most of the existing classes from both the engine and game package. The new classes created are inside the package “game.items” and also the NurseJoy class. The role of the new items classes is to hold the unique and similar properties of each item, and allow the items to be initialised into the game world. The NurseJoy class is responsible for holding the transaction of “candies” in exchange for other items. The new classes in game.items extend the abstract “Item” class, as they share some common properties and the use of repeated code is kept to a minimum. The items also interact with the Player class which extends the abstract “Actor” class. The relationships that exist between the Actor class and game.items classes are there to show that the items have an effect on the Pokémon, who are also Actors. A separate class for Pokémon could be created to make this clearer and help obey the Single Responsibility Principle as currently the Actor class is responsible for many different actors. However, a separate class for Pokémon was not included as this would make the diagram more complex and harder to read. The actions for dropping and picking up the items are also shown, and the abstract Ground class is included as this will hold the items that are dropped.

Requirement 4:



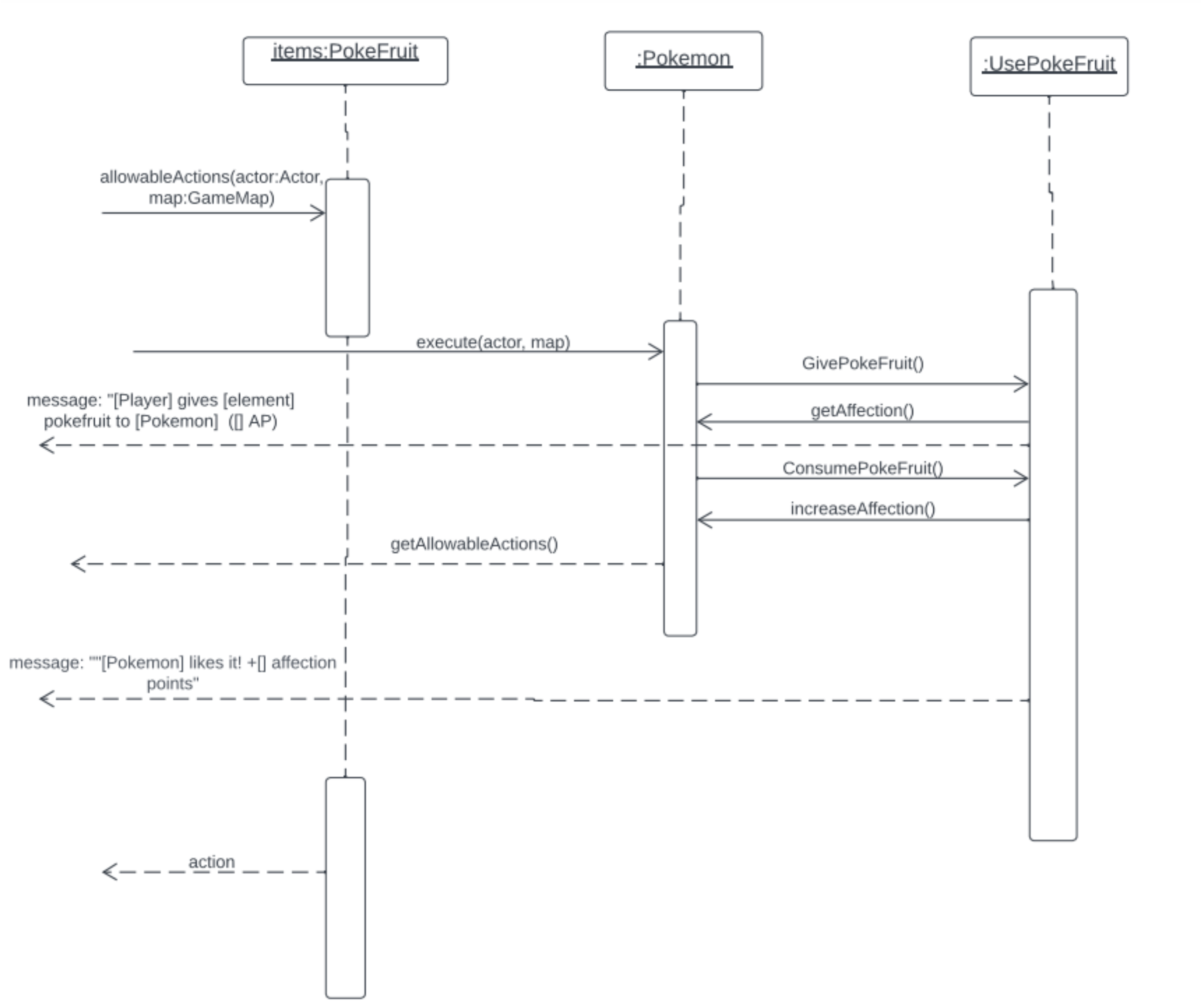
UML Requirement 4 Rationale:

This requirement is about using the Pokémon's affection for the player in order to determine if the Pokémon can/cannot be captured and if it will follow the player around (behaviours of the actor). The requirement also determines if the Pokémon's affection increases, or decreases based on fruits given to them and their element. This affection point system is managed by the Affection Manager class which would store the Pokémon with their respective affection points. Furthermore, the status Enum is also utilised in this design as this makes it easy to know which Pokémon are catchable and which are not and can be changed based off affection points. This Enum ensures that code doesn't use an excessive number of literals and follows the open-closed principle by keeping the code much more concise and making it, so we won't have to code a various range of conditional for checks.

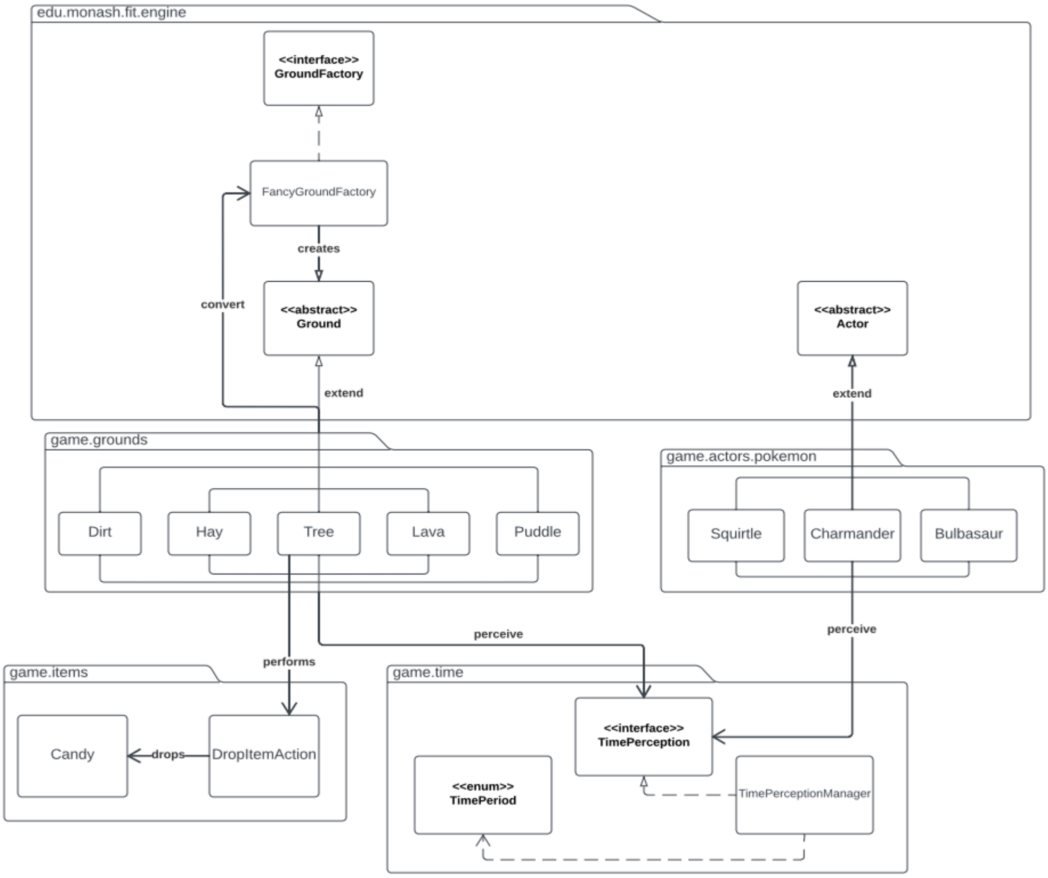
We also create a ConsumePokeFruit action class that acts as a subclass for Action and is used to program what is done if the poke fruit is consumed by Pokémon. Instead of consuming the poke fruit directly in the Items subclass, we regard the intended usage of the engine and follow the single responsibility principle in this scenario by reflecting that most of the functionalities are executed after the poke fruit is given to the Pokémon.

The given Sequence Diagram outlines the process of a ConsumePokeFruit by a Pokémon to increase affection:

Interactive Diagram:


items:PokeFruit
:Pokemon
:UsePokeFruit
allowableActions(actor:Actor, map:GameMap)
execute(actor, map)
GivePokeFruit()
message: "[Player] gives [element] pokefruit to [Pokemon] ([] AP)
getAffection()
ConsumePokeFruit()
increaseAffection()
getAllowableActions()
message: ""[Pokemon] likes it! +[] affection points"
action

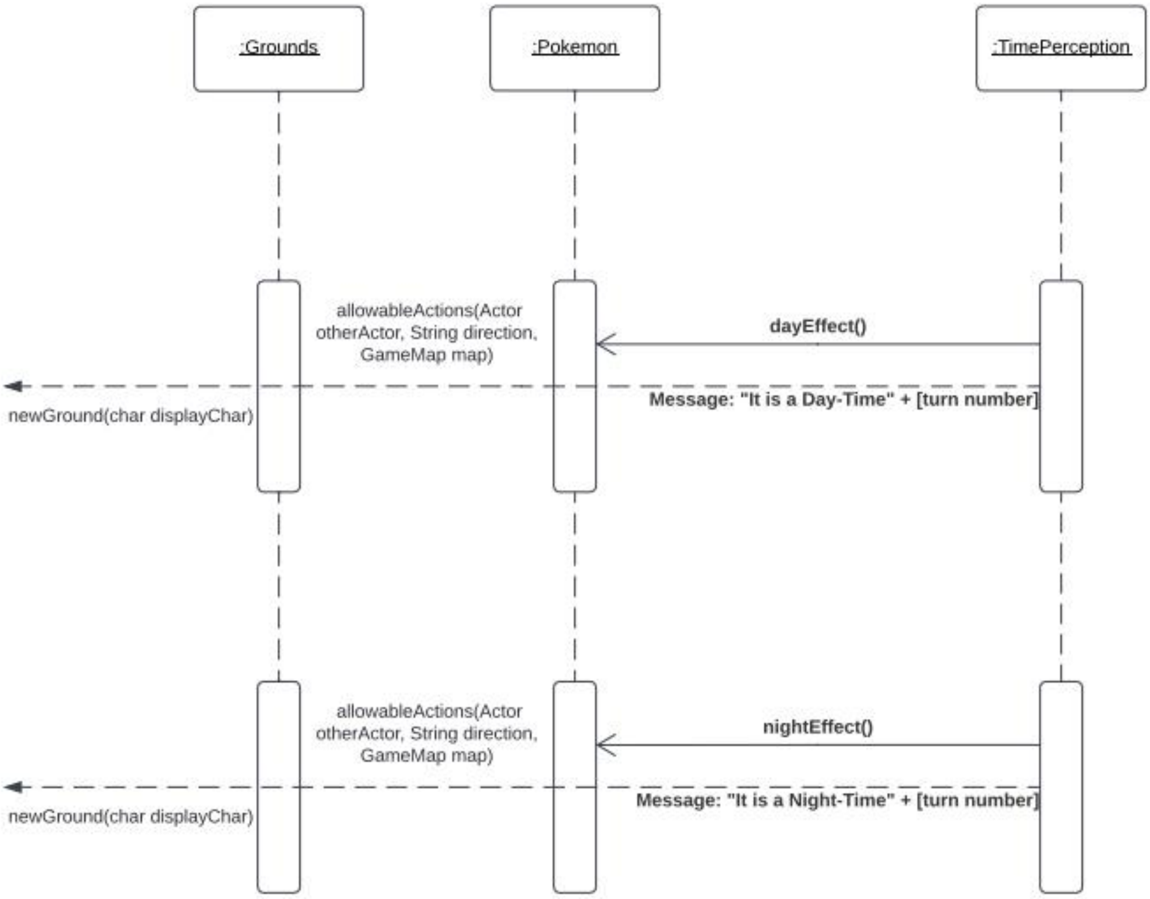
Requirement 5:



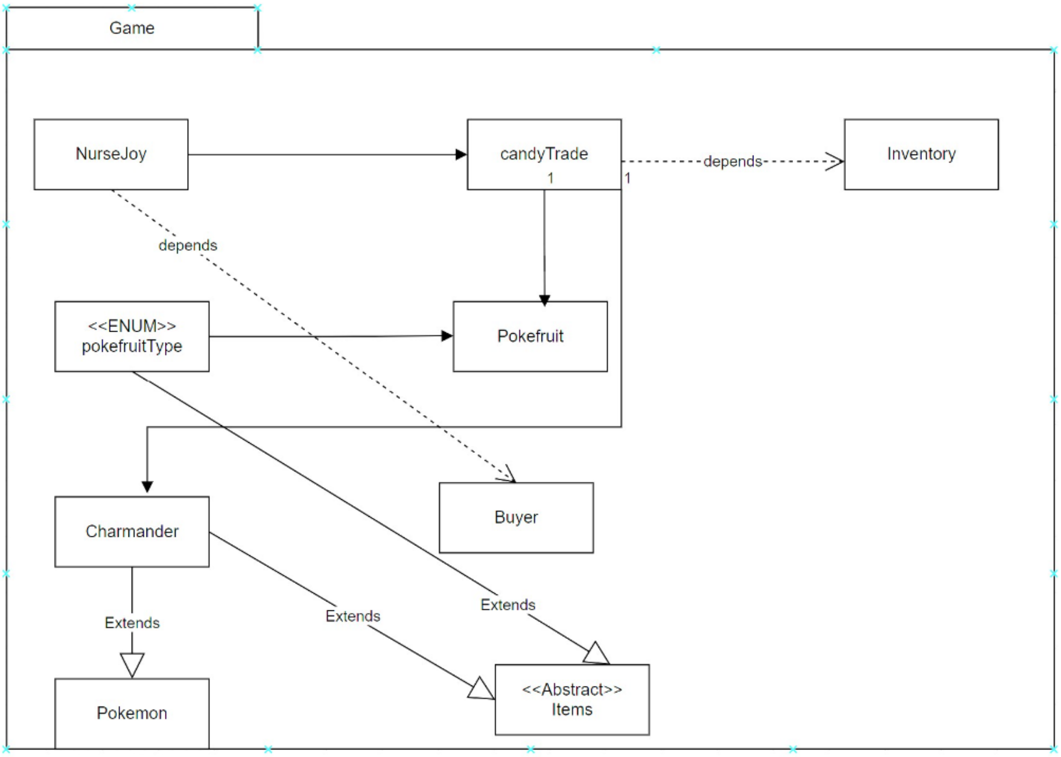
UML Requirement 5 Rationale:

This diagram represents how the classes inside the game.grounds package and the new game.actors.pokemon classes perceive time and as a result experience changes. The classes in the game.grounds package based on what they perceive from the TimePerception interface, (which determines the time of day using the other two game.time classes) convert themselves into their respective ground type using the FancyGroundFactory class, which creates the new Ground objects using the abstract Ground class. The Ground class is made abstract to avoid the repetition of code (DRY). The Tree class also has a chance of dropping a candy, which it does so using the DropItemAction class. The Pokemon which extend the Actor class also perceive time from the TimePerception interface, which would then apply the statuses of either restoring their hit points or taking damage. It was decided the classes that apply these statuses where not included in this diagram as these behaviours and actions are complex enough to require their own UML diagram.

Interactive Diagram:


:Grounds
:Pokemon
:TimePerception
allowableActions(Actor otherActor, String direction, GameMap map)
dayEffect()
newGround(char displayChar)
Message: "It is a Day-Time" + [turn number]
allowableActions(Actor otherActor, String direction, GameMap map)
nightEffect()
newGround(char displayChar)
Message: "It is a Night-Time" + [turn number]

Requirement 6:



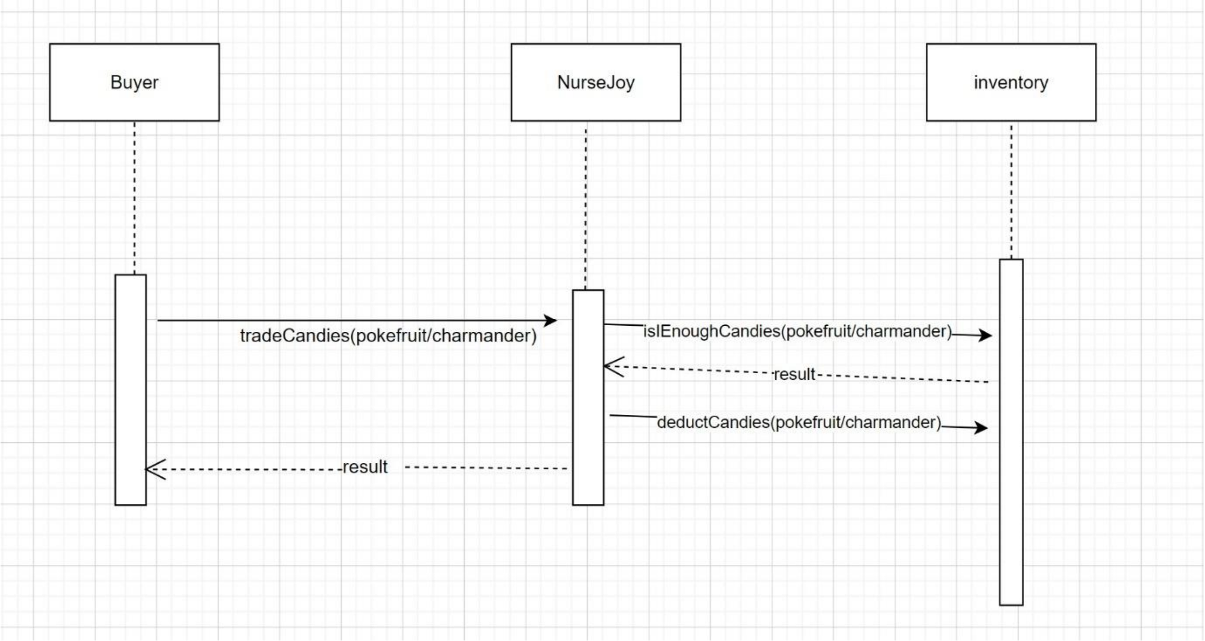
UML Requirement 6 Rationale:

In this requirement, the class NurseJoy depends on the buyer i.e. the player that whether they want to swap items for candies. The nurseJoy inherits candyTrade that depends on inventory for the number of candies and also associates classes Pokefruit and Charmander for a trade. The Enum class Pokefruit consists of water pokefruit, fire Pokefruit, and grass pokefruit, for an eligible exchange.

Items is an abstract class and class Charmander and PokeFruit inherits this class.

Charmander is a Pokémon so it inherits all its properties from the Pokémon class.

Interactive Diagram:


Buyer
NurseJoy
inventory
tradeCandies(pokefruit/charmander)
isIEnoughCandies(pokefruit/charmander)
result
deductCandies(pokefruit/charmander)
result