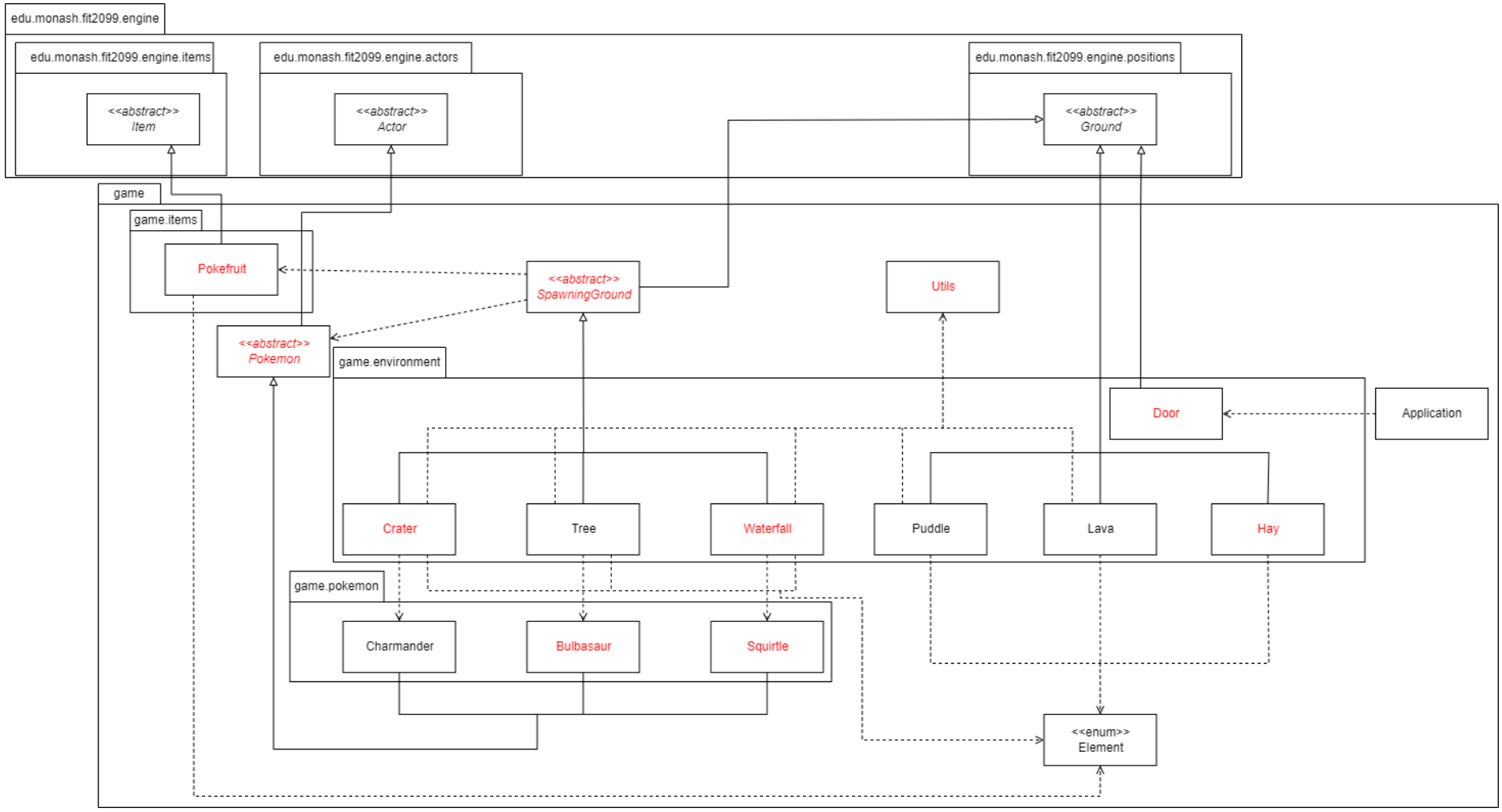
Assignment 3
FIT2099
Object Oriented Design and Implementation

Team members:

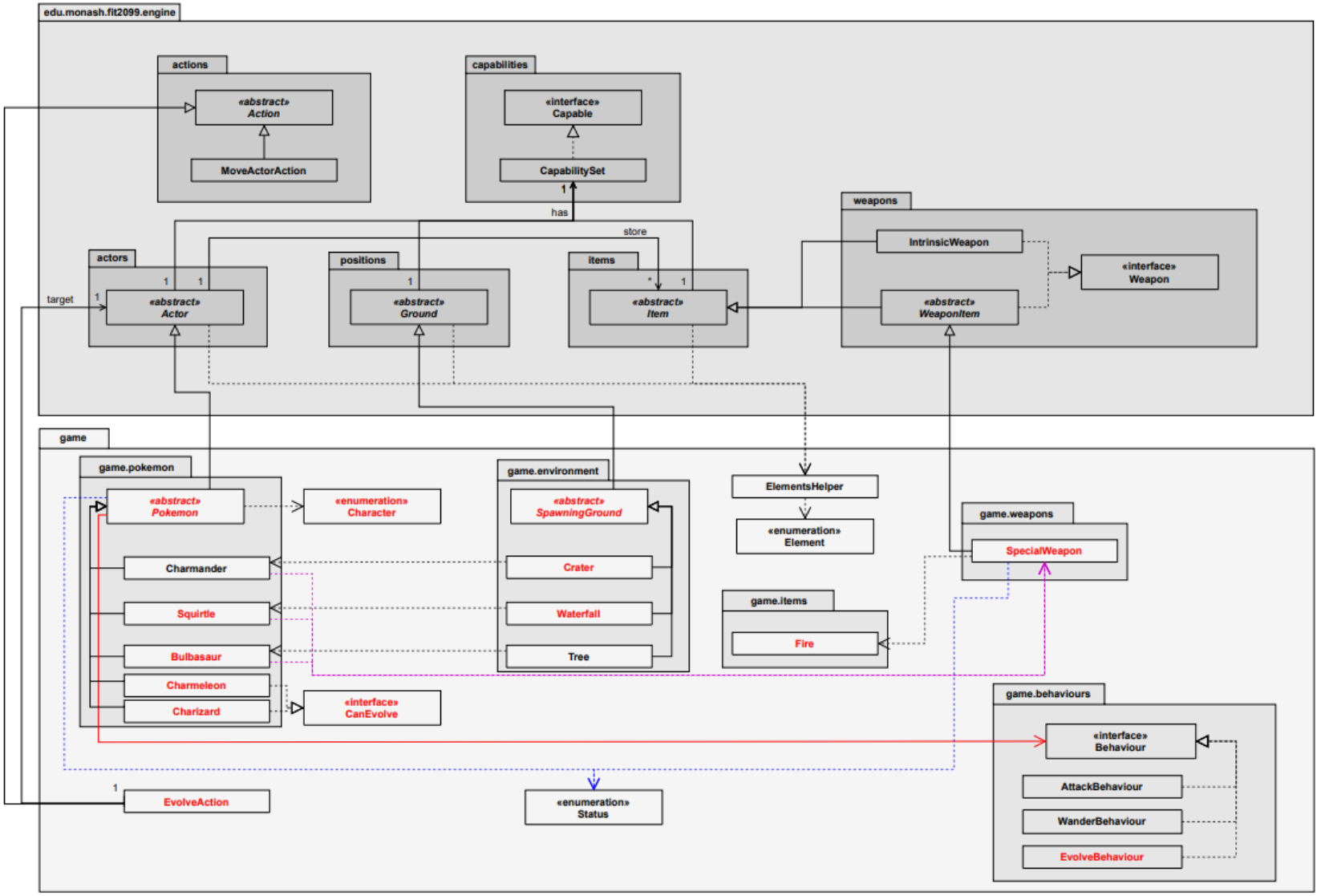
1. Fong Zhiwei (32686072)
2. Soh Meng Jienq (30531608)
3. Leong Xin Yun (32835477)

1 UML Diagrams

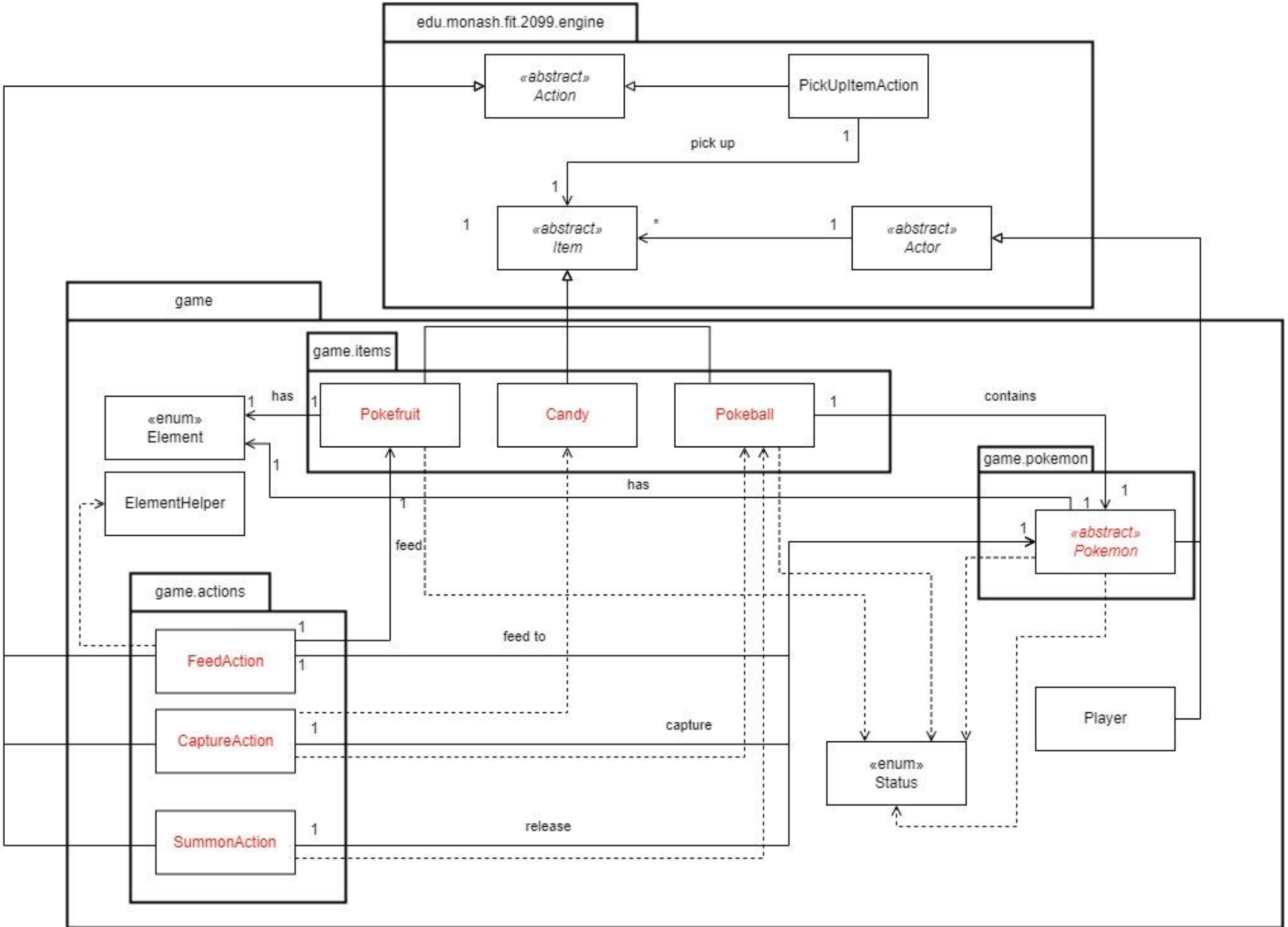
1.1 Environment



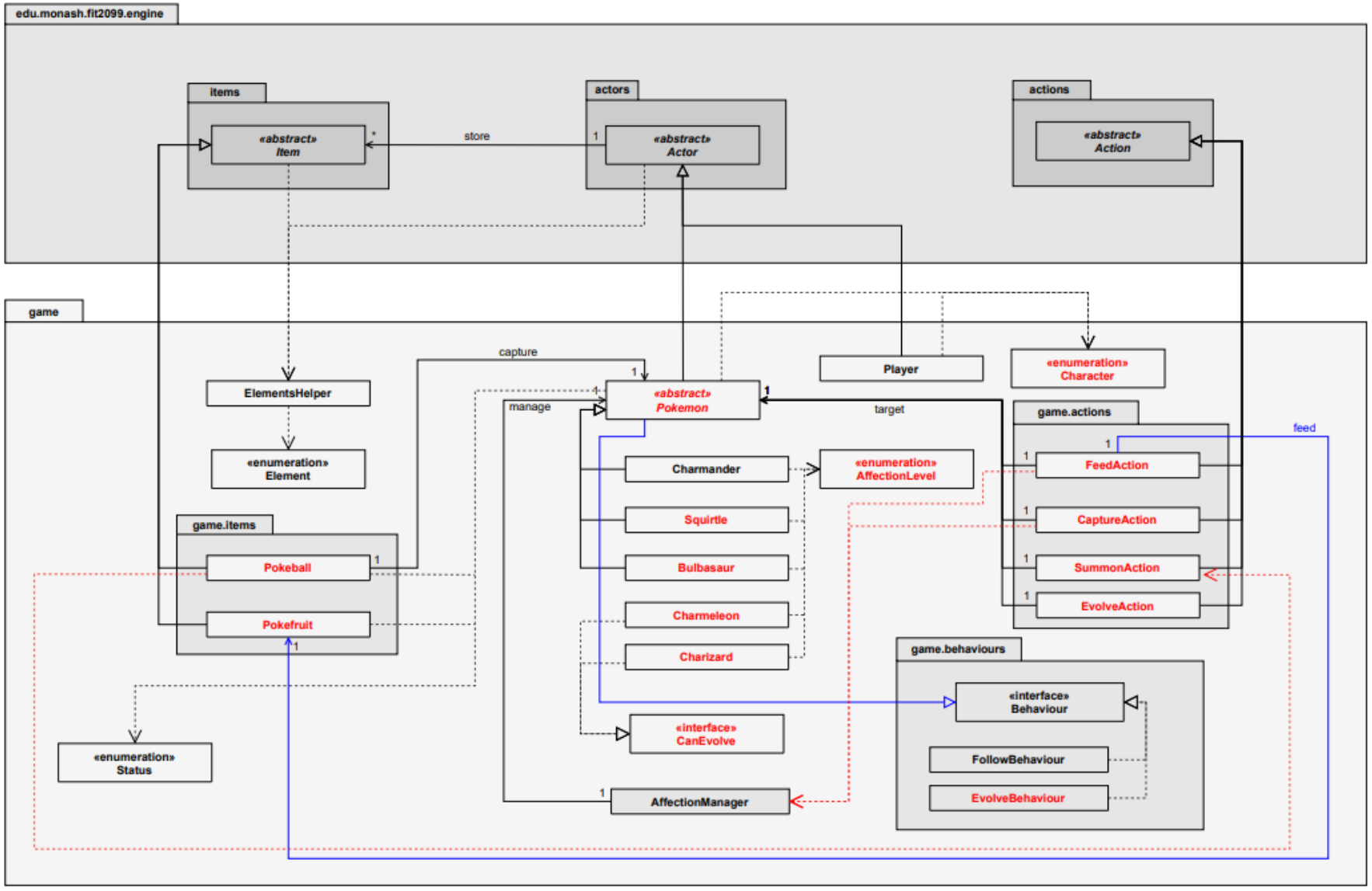
1.2 Pokemons


edu.monash.fit2099.engine
actions
«abstract»
Action
MoveActorAction
capabilities
«interface»
Capable
CapabilitySet
1
has
store
weapons
IntrinsicWeapon
«interface»
Weapon
«abstract»
WeaponItem
actors
1
1
target
1
«abstract»
Actor
positions
1
«abstract»
Ground
items
*
1
«abstract»
Item
game
game.pokemon
«abstract»
Pokemon
«enumeration»
Character
Charmander
Squirtle
Bulbasaur
Charmeleon
Charizard
«interface»
CanEvolve
game.environment
«abstract»
SpawningGround
Crater
Waterfall
Tree
ElementsHelper
«enumeration»
Element
game.weapons
SpecialWeapon
game.items
Fire
1
EvolveAction
«enumeration»
Status
game.behaviours
«interface»
Behaviour
AttackBehaviour
WanderBehaviour
EvolveBehaviour

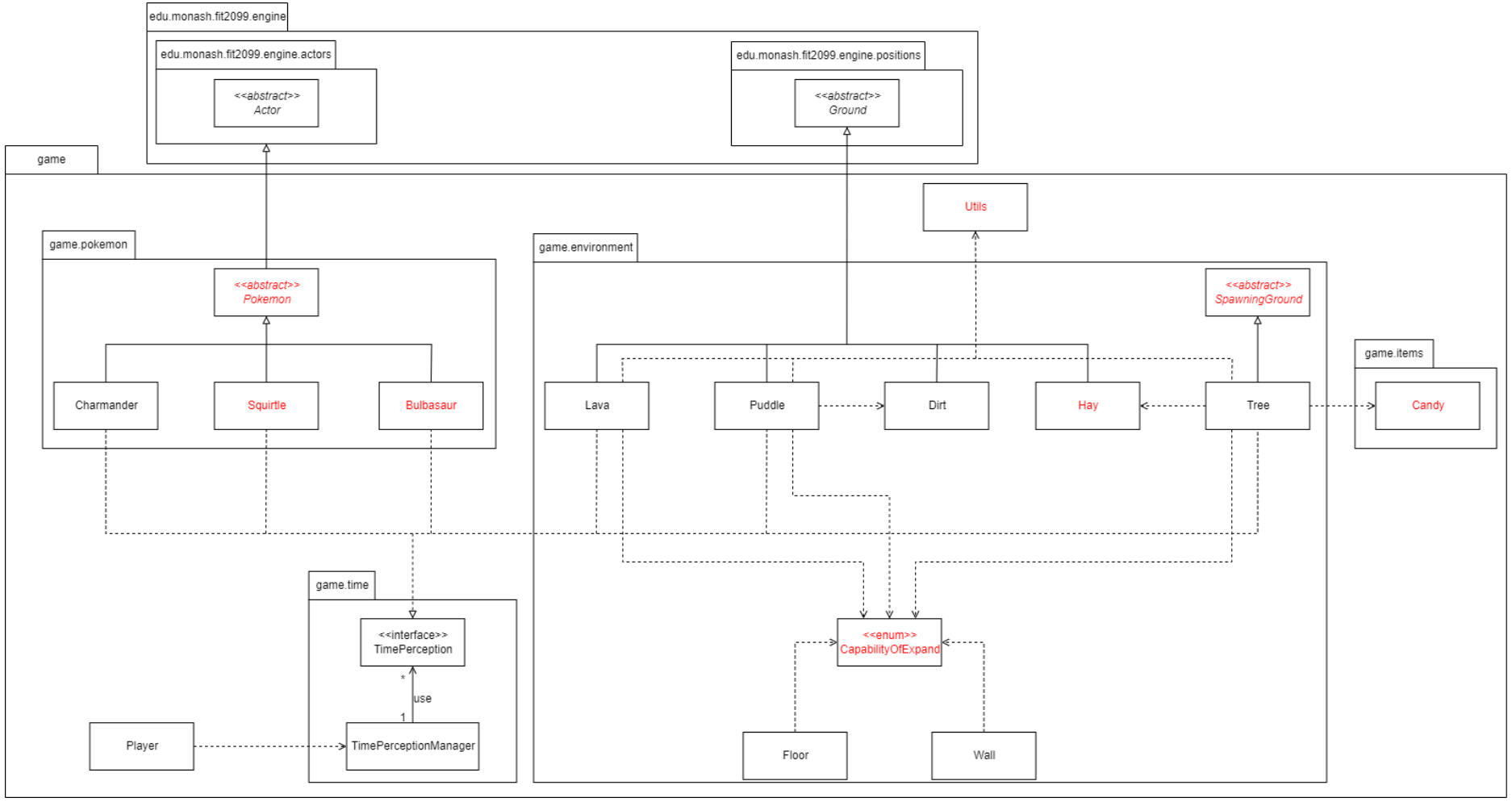
1.3 Item


edu.monash.fit.2099.engine
«abstract»
Action
PickUpItemAction
pick up
1
1
1
«abstract»
Item
*
1
«abstract»
Actor
game
game.items
Pokefruit
Candy
Pokeball
1
contains
has
1
1
«enum»
Element
1
ElementHelper
has
1
1
1
game.pokemon
1
1
«abstract»
Pokemon
feed
game.actions
1
feed to
FeedAction
1
CaptureAction
1
capture
SummonAction
1
release
«enum»
Status
Player

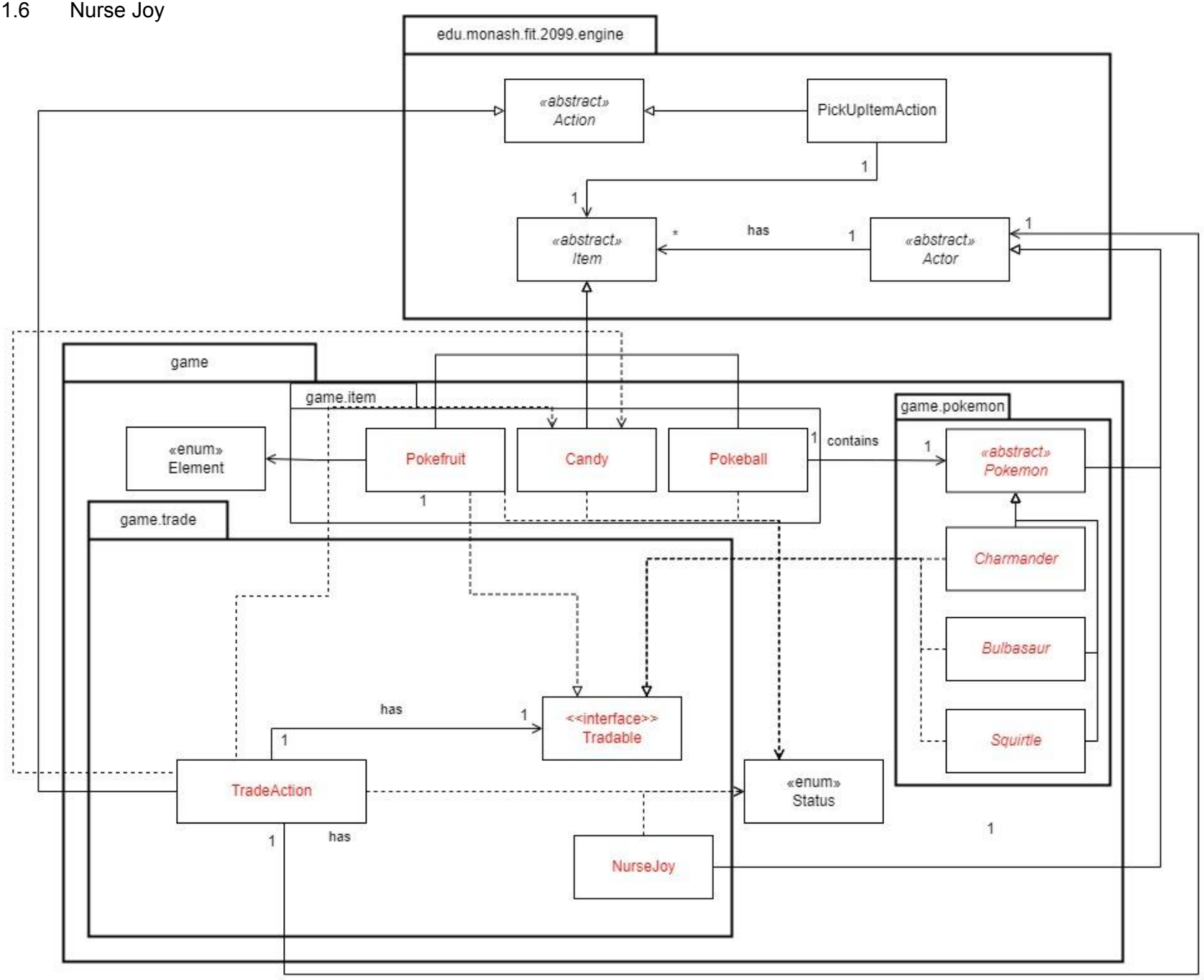
1.4 Interactions



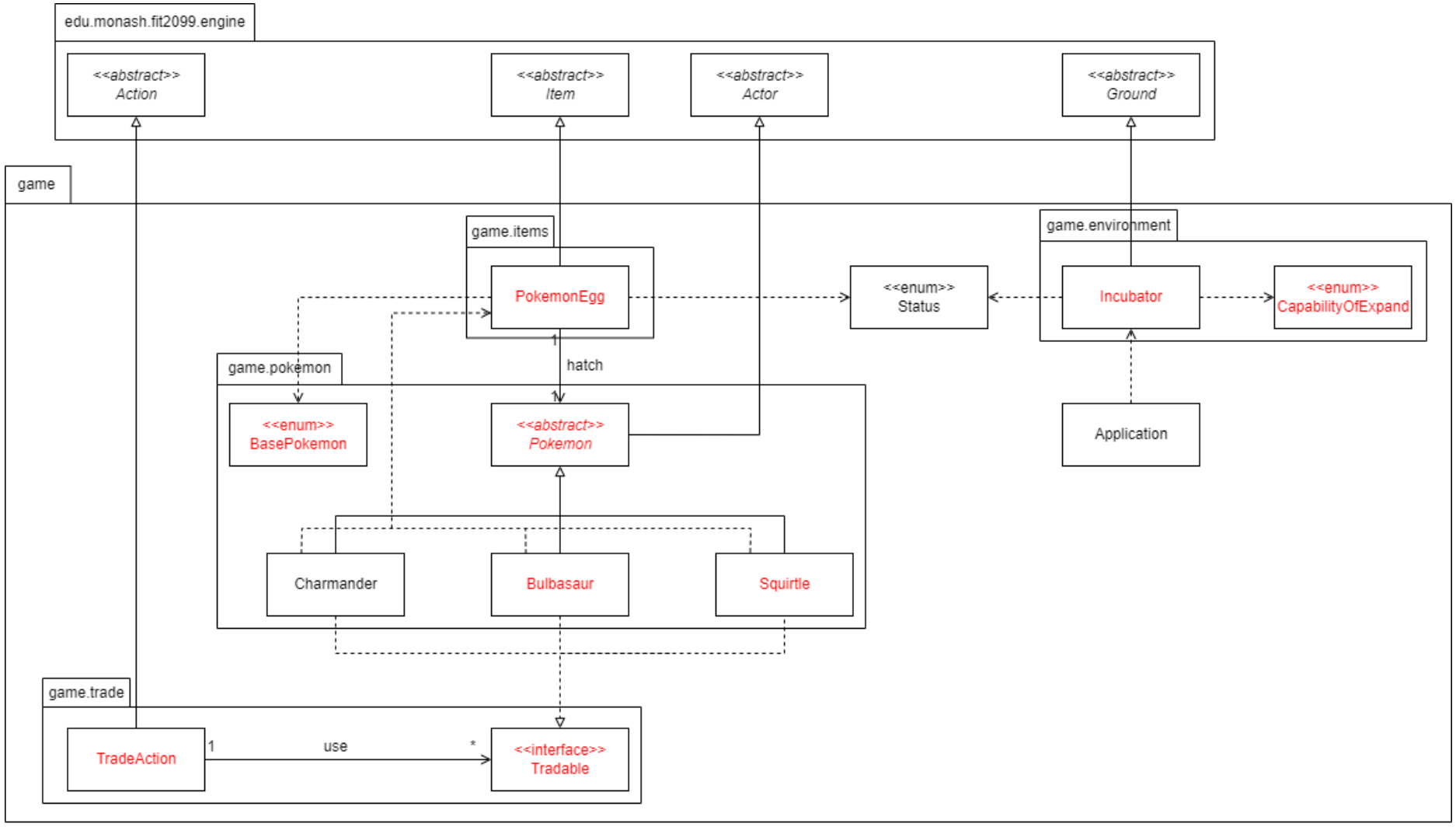
1.5 Day and Night



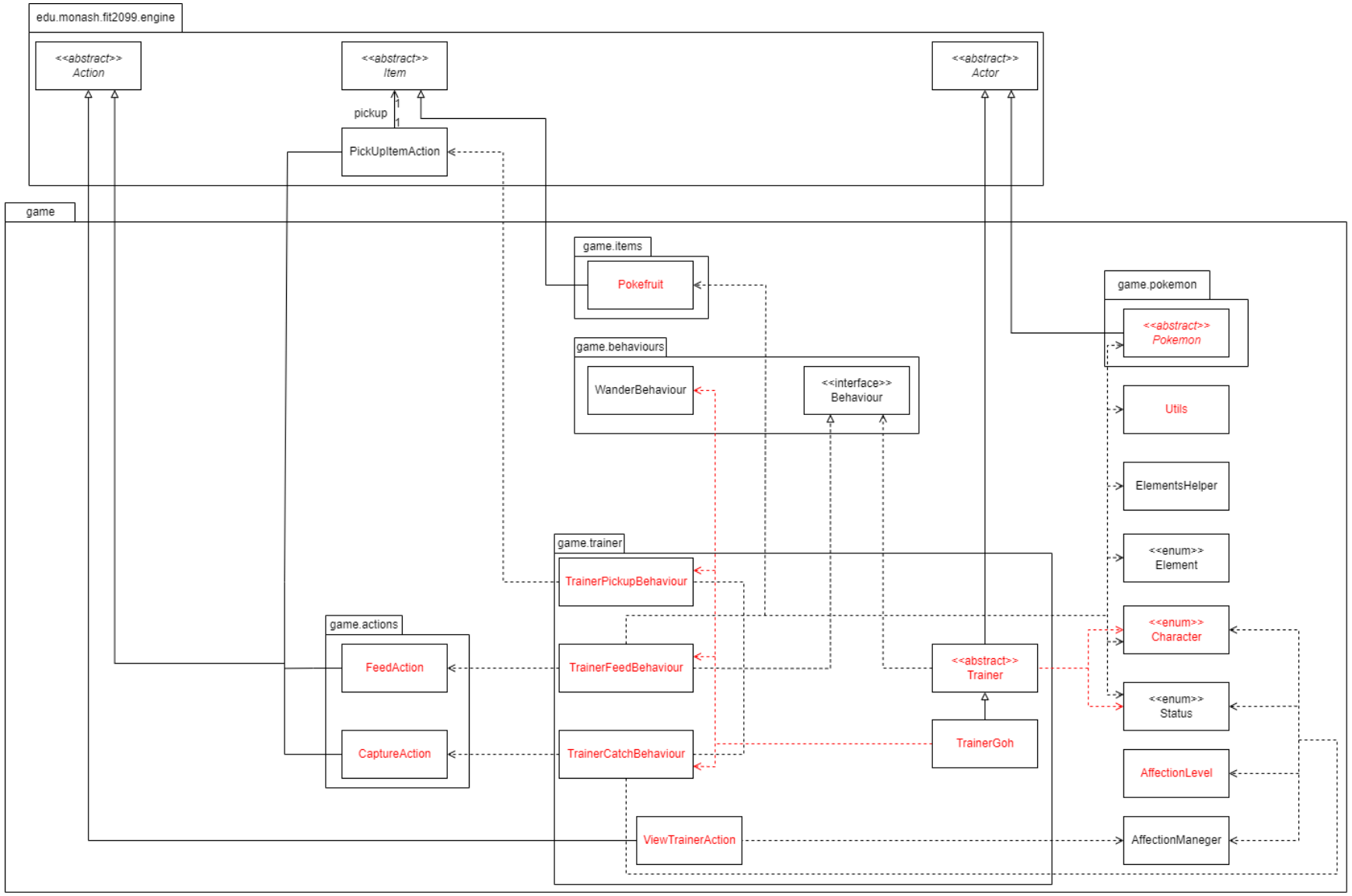
1.6 Nurse Joy


edu.monash.fit.2099.engine
«abstract»
Action
PickUpItemAction
1
1
«abstract»
Item
*
has
1
«abstract»
Actor
1
game
game.item
game.pokemon
«enum»
Element
Pokefruit
Candy
Pokeball
1
contains
1
«abstract»
Pokemon
1
game.trade
Charmander
Bulbasaur
Squirtle
has
1
<<interface>>
Tradable
1
TradeAction
«enum»
Status
1
has
1
NurseJoy

1.7 Pokemon Egg and Incubator



1.8 New Trainer



2 Design rationale

Highlighted part is the deleted part, and the part with purple words is newly added.

2.1 Environment

SpawningGround extends Ground. Abstract SpawningGround class that inherits abstract Ground class has the necessary methods to generate terrain. SpawningGround class is similar to Ground class, the difference is that it is a dynamic ground. And, the reason to make SpawningGround to be an abstract class is that abstract class is useful to create new types of terrain class, like Crater class, Tree class and Waterfall class.

Crater extends SpawningGround, **Tree extends SpawningGround**, **Waterfall extends SpawningGround.** They are all dynamic ground, hence they inherit from SpawningGround.

Crater---<<create>>--->Charmander, **Tree---<<create>>--->Bulbasaur** and **Waterfall---<<create>>--->Squirtle**. These 3 types of terrain will spawn 3 different types of pokemons. And, the terrains don't need to change any information, like behaviours and capabilities of pokemons. Thus, using the dependency relationship is enough for them. It is also an alternative to have an association relationship between the three terrains and the three pokemons. In other words, pokemons will be the attributes of the terrains, and the capabilities of the pokemons can be modified in terrain classes. However, it is better to reduce the dependency between classes to prevent high coupling.

Charmander extends Pokemon, Bulbasaur extends Pokemon, Squirtle extends Pokemon. These 3 types of pokemon inherit abstract Pokemon class, because it contains the behaviour of a pokemon.

SpawningGround---<<create>>--->Pokemon and **SpawningGround---<<create>>--->Pokefruit.** SpawningGround abstract class has Pokemon abstract class and Pokefruit concrete class inside. This is because SpawningGround always comes with these two objects. It is a place to create a pokemon and drop fruit at every turn. The reason to make Pokemon an abstract class, and a Pokefruit concrete class is that there are many pokemon classes, like Charmander, Bulbasaur, Squirtle, but Pokefruit only needs one class with a single enum element difference. It is also possible to make Pokefruit as an abstract class, but it needs to generate three more

concrete classes to inherit the Pokefruit abstract class. Hence, it is better to set Pokefruit as a concrete class. Both Pokemon abstract class and Pokefruit concrete class are aligned with the Open-closed Principle.

SpawningGround---<<create>>--->Pokefruit and **Pokefruit---<<use>>--->Element.** SpawningGround (Crater, Tree, Waterfall) drops pokefruit according to its location. And, each pokefruit has a different element (Fire, Water, Grass). From the game map, replace the SpawningGround with Pokefruit. Both of the relationships are dependency, as it only requires reading the information from the classes.

Crater---<<use>>--->Element, Tree---<<use>>--->Element, Waterfall---<<use>>--->Element. These three types of SpawningGround have different elements (Fire, Water, Grass). Hence, each terrain has a dependency relationship with Element is a better choice. The reason for having the dependency relationship with Element is that these 3 terrain only requires reading the information from the enumeration.

Puddle extends Ground, Lava extends Ground, Hay extends Ground. Puddle, Lava and Hay concrete classes inherit the abstract Ground class which has the elementary methods to form a type of terrain with its respective capabilities. This has aligned with the Dependency Inversion Principle, as Ground abstract class does not depend on Puddle, Lava and Hay concrete classes.

Hay---<<use>>--->Element, Lava---<<use>>--->Element, Puddle---<<use>>--->Element. These three types of Ground have different types of elements. And, dependency relationship with Element enumeration is enough, as these 3 terrain classes only require reading the information of Element enumeration.

Crater---<<use>>--->Utils, Tree---<<use>>--->Utils, Waterfall---<<use>>--->Utils, Puddle---<<use>>--->Utils, Lava---<<use>>--->Utils. All these terrains use Utils Class to check the probability percentage. And, they only need to read the Utils class, and not need to modify the class. Thus, a dependency relationship is enough.

Door extends Ground. Door concrete class inherits the abstract ground class which has the essential methods to form a ground. Hence, the door can use the method in ground class and add a special action to the action list. And, the special action is to link the game map to Pokemon Center.

2.2 Pokemons

The Pokemons UML diagram showcases an object oriented system for a rogue-like pokemon game that has new characters, grounds and weapons added into the existing engine and game packages. New classes are colored in blue, whereas lines in red are only for visual purposes.

SpawningGround extends Ground. SpawningGround is an abstract class that extends the abstract Ground class. This class is made abstract separately to all other normal grounds (Lava, Puddle and Hay) as they have their own set of attributes and methods (e.g. ability to spawn pokemons), thus eliminating redundant attributes and methods for the spawning grounds (Crater, Waterfall and Tree). The final design included SpawningGround as an abstract class. The abstract class is a restricted class that cannot be instantiated, making the final design aligned with the Open-Closed Principle because it provides flexibility to support future alterations.

Crater extends SpawningGround, Waterfall extends SpawningGround, Tree extends SpawningGround. Crater, Waterfall and Tree are concrete classes extending the abstract SpawningGround class, where only these 3 types of ground are able to spawn Pokemons. These classes are extended separately as they have their own capabilities of spawning a pokemon (e.g. Crater is only capable of spawning Charmander). Alternatively, these subclasses could be merged into a single subclass. However, doing so would have populated the objects with unrelated behaviours and functionalities, not only it would be difficult to maintain but it could also affect other classes that are closely related to it. Thus, this alternative was discarded. The final design is aligned with the Single Responsibility Principle as all extensions of the class could have their own unique attributes and behaviour.

Pokemon extends Actor. Pokemon class extended the abstract Actor class; they share similar attributes. Pokemon is created as an abstract class as there are different types of Pokemons in the game, namely Charmander, Squirtle and Bulbasaur, which all share some common attributes and methods. Extending them from an abstract class eliminates repetitions, thus supporting the Don’t Repeat Yourself (DRY) principle. The design is also aligned with the Open-Closed Principle as it allows class extensions without mutating the existing base code.

Charmander extends Pokemon, Squirtle extends Pokemon, Bulbasaur extends Pokemon. Charmander, Squirtle and Bulbasaur are concrete classes that extend the abstract Pokemon class as they share a set of common attributes and methods. These classes extended separately to preserve their uniqueness (e.g. having different names, capabilities, or even additional methods to themselves). Alternatively, these subclasses could have extended the abstract Actor class instead of the Pokemon class. However, this is not as effective when there are modifications towards all the pokemon because changes would need to be made to all the pokemon classes. Thus, this alternative was rejected. The final design is aligned with the Single Responsibility Principle as all the class extensions have their own unique functionalities and responsibilities. It also supports the DRY principle as modification to the pokemons' common attributes and methods could be made directly to the abstract class. This would save a lot of time and effort when there are necessary changes to be made to all pokemons.

Charmander---<<implement>>--->Behaviour, Squirtle---<<implement>>--->Behaviour, Bulbasaur---<<implement>>--->Behaviour. Charmander, Squirtle and Bulbasaur implement Behaviour interface. The pokemons implement the behaviour in deciding which action to perform next. It can also be used to create new actions, thus each pokemon would have their own unique actions. Alternatively, the pokemons could have a direct association relationship with the behaviour concrete classes (FollowBehaviour, WanderBehaviour and AttackBehaviour). However, it is not recommended to do so as a concrete class should not depend on another concrete class. Thus, this implementation was discarded. The final design involves pokemon subclasses to implement the behaviour interface class, which is aligned with the Dependency Inversion Principle because all the pokemon concrete subclasses depend on the behaviour interface (abstraction).

Crater---<<create>>--->Charmander, Waterfall---<<create>>--->Squirtle, Tree---<<create>>--->Bulbasaur. Crater has a dependency on Charmander; Waterfall has a dependency on Squirtle; Tree has a dependency on Bulbasaur. As the type of pokemons spawned depends on the type of ground, it is logical to have dependency relationships between these classes.

Pokemon---<<use>>--->Status. Pokemon have a dependency on Status enumeration. Status is an existing enum class in the game package and it is used to determine the current state of the actor or item. Some enum types are added into the enum class to further differentiate between the actors and items. E.g. CATCHABLE is used to identify if the actor is catchable, FOOD is used to identify if an item is fedable to the pokemon (i.e. pokefruit). The enum class has dependency relationships with all the items and actors in the game as the current state of the instance changes with their capabilities and actions.

Pokemon---<<use>>--->Character. Pokemon has a dependency on Character enumeration. Character is created as an enum class because there are only two types of

characters inside the game, which are player and non-player character (NPC) and it should not be mutable throughout the game. The enum class has dependency relationships with all the actors in the game. This is because different types of characters have different sets of actions and behaviours to be executed.

Charmander---<<use>>--->SpecialWeapon, Bulbasaur---<<use>>--->SpecialWeapon, Squirtle---<<use>>--->SpecialWeapon. Charmander, Bulbasaur and Squirtle have a dependency on the SpecialWeapon class. SpecialWeapon is created as a concrete class and used to create a special weapon object. Ember, Bubble and VineWhip are replaced with the SpecialWeapon class. The reason being is that these classes are redundant as the functionality of the class is to only create a special weapon object for the pokemon. The class can be used to create different weapons by initialising specific values when calling the constructor of the class. SpecialAttack enumeration is also removed from the design as the verbs of each weapon are taken in as a parameter by the constructor whenever a weapon object is created. Charmander, Bulbasaur and Squirtle have a direct relationship with the SpecialWeapon class as each of them are only entitled to a specific weapon when a special attack is granted on that particular turn.

Charmeleon extends Pokemon, Charizard extends Pokemon. Charmeleon and Charizard are concrete classes that extend the abstract Pokemon class as they share a set of common attributes and methods. These classes extended separately so as to preserve their uniqueness. Alternatively, these subclasses could have extended the Charmander class instead of the Pokemon class since they are the evolved version of the Charmander. However, this is not as effective because Charmeleon and Charizard do not share a high similarity of attributes and methods. Most of the methods have different levels of complexity, thus it would be easier to implement them separately. Moreover, it does not make sense to have the Charmander as an abstract class when Charmander Pokemon exist as an instance inside the game. Thus, this alternative was rejected. The final design is aligned with the Single Responsibility Principle as all the class extensions have their own unique functionalities and responsibilities.

Charmeleon---<<use>>--->SpecialWeapon, Charizard---<<use>>--->SpecialWeapon. Charmeleon and Charizard have a dependency on the SpecialWeapon class. SpecialWeapon is created as a concrete class and used to create a special weapon object. Weapon instances are created using the SpecialWeapon class instead of solely creating a concrete class extending an abstract SpecialWeapon class for each particular weapon. The reason being is that these classes are redundant as the functionality of the class is to only create a special weapon object for the pokemon. The class can be used to create different weapons by initialising specific values when calling the constructor of the class. Charmeleon and Charizard have a direct relationship with the SpecialWeapon class as each of them are entitled to a different set of weapons when a special attack is granted on that particular turn.

Charmander---<<has>>--->Behaviour, Squirtle---<<has>>--->Behaviour, Bulbasaur---<<has>>--->Behaviour, Charmeleon---<<has>>--->Behaviour, Charizard---<<has>>--->Behaviour. Charmander, Squirtle, Bulbasaur, Charmeleon and Charizard each have a map of behaviours. The pokemons implements the behaviour in deciding which action to perform next. It is also possible to create new actions in that behaviour category, thus each of them could have their own unique actions. Alternatively, the pokemons would have shared a single map of behaviours. However it is not recommended as not all actions are allowable for every pokemon (e.g. EvolveBehaviour only works for Charmander and Charmeleon). Besides that, the priority of actions will also differ based on the type of pokemon. For instance, EvolveBehaviour will have top priority over other actions for Charmander and Charmeleon, whereas other pokemon will have FollowBehaviour as their top priority. Thus, this idea was discarded. The final design implementation is to create a map of behaviours for each pokemon instance so that all pokemon could behave differently.

Charmander---<<implement>>--->CanEvolve, Charmeleon---<<implement>>--->CanEvolve. Charmander and Charmeleon implements CanEvolve interface for evolving purposes. CanEvolve is a new interface created with an abstract method inside to check if the pokemon meets the criteria to perform the evolve action. Charmander and Charmeleon implements this interface in deciding whether to add the EvolveBehaviour in their behaviour list. An alternative way is to perform the validation in the Pokemon abstract class. However, this is not recommended because not all pokemon are able to evolve, thus the validation process would be redundant for the other pokemon. This implementation was discarded and the final design implementation is to create an interface that checks the validity of the pokemon to be evolved.

2.3 Item

Pokeball extends Item. Pokeball concrete class inherits Item abstract class. Pokeball has an attribute of Pokemon type because each Pokeball will contain a single Pokemon. There is an infinite number of Pokeball CaptureAction will instantiate a Pokeball with the corresponding target of the CaptureAction if CaptureAction is successful.

CaptureAction---<<create>>--->Pokeball. CaptureAction will instantiate a Pokeball with the corresponding Pokemon of the CaptureAction. The Pokeball with the Pokemon will be instantly added to the Player's inventory (ArrayList of item) to prevent any other Actors from picking up the Pokeball.

SummonAction---<<use>>--->Pokeball. ReleaseAction will remove a Pokeball with the corresponding Pokemon from the Player's Inventory. The Pokeball will not be dropped to prevent other Actors from picking up the Pokeball.

Pokeball---<<contains>>--->Pokemon. Each instance of Pokeball will have an instance of Pokemon inside it. Pokeball has an association relationship with Pokemon.

Pokefruit extends Item implements Tradable. Pokefruit concrete class inherits Item abstract class. Pokefruit has an attribute of Element which is a Enum to determine which type of Pokefruit as there are 3 types of Pokefruit which is Fire Pokefruit, Water Pokefruit and Grass Pokefruit. Player will be able to pick up or drop the Item because of the inherited method from Item abstract class

The tradable interface that pokefruit implements contains a function called tradedWith to calculate the price of the Pokefruit object and if the Player has enough candies and also responsible for removing the Candy from the Player's inventory if the Player has enough candies.

Charmander implements Tradable.

The tradable interface that Charmander implements contains a function called tradedWith to calculate the price of the Charmander object and if the Player has enough candies and also responsible for removing the Candy from the Player's inventory if the Player has enough candies.

FeedAction---<<removes>>--->Candy. FeedAction removes the selected pokefruit from the Player's inventory. FeedAction knows about Candy.

Candy extends Item. Candy concrete class inherits Item abstract class. Candy will be instantiated upon a successful CaptureAction and dropped to the ground at the location where CaptureAction succeeded. Players will be able to pick up or drop the Candy because of the inherited method from Item abstract class.

CaptureAction---<<create>>--->Candy. CaptureAction will instantiate a single Candy and drop it at the location of the CaptureAction. There is a dependency relationship between them.

Pokefruit---<<use>>--->Status, Pokeball---<<use>>--->Status, Candy---<<use>>--->Status, The three items has the Status enum as their capability so methods involving these 3 items can use the items's capability to know which type of item they are. This adheres to the Open-closed Principle and Liskov Substitution Principle as all of the items can add different statuses in their respective subclass but an Item can know which specific item they are by checking their capabilities.

The tradable interface the Pokefruit and Charmander implements also adheres to the Dependency inversion principle such that the tradeAction of these objects does not rely on the prices being stored in the Item or Nursejoy. The methods inherited by these objects already implement the functionality.

While designing the Items, the Pokeball, Pokefruit and Candy all inherit from the parent abstract Item class. This is to allow every class that inherits Actor abstract class to be able to store these items with the ArrayList of items inherited in Actor abstract class. This satisfies the Open-closed Principle as Pokeball, Pokefruit and Candy all can be extended without having to modify the Item class and they can have their own details in their subclasses.

If we tried to create Pokefruit, Pokeball and Candy as standalone classes, it would violate the Open-closed principle. This is because the Actor abstract class that contains an ArrayList of Items as it attributes needs to be modified with a separate list for each item when extending to its subclasses as a list of items cannot be used in this case.

2.4 Interactions

The Interactions UML diagram showcases an object oriented system for a rogue-like pokemon game that has new characters, items and actions added into the existing engine and game packages. New classes are colored in blue, whereas lines in red are only for visual purposes.

Pokemon extends Actor. Refer to Pokemons.

Charmander extends Pokemon, Squirtle extends Pokemon, Bulbasaur extends Pokemon. Refer to Pokemons.

Charmander---<<use>>--->AffectionLevel, Squirtle---<<use>>--->AffectionLevel, Bulbasaur---<<use>>---> AffectionLevel. Charmander, Squirtle and Bulbasaur have a dependency on AffectionLevel enumeration. AffectionLevel is created as an enum class as each affection level has a constant range of affection points and should not be modified throughout the game. The enum class has dependency relationships with all the pokemons. The reason is that the pokemons' behaviour towards the player (its trainer) depends on its affection level. The way pokemons treat its trainer changes with its affection level towards the trainer.

Charmander---<<implement>>--->Behaviour, Squirtle---<<implement>>--->Behaviour, Bulbasaur---<<implement>>---> Behaviour. Refer to Pokemons.

Pokeball extends Item, Pokefruit extends Item. Pokeball and Pokefruit are concrete classes extending the abstract Item class, a base class that contains a set of common attributes and methods for all item objects. These classes are extended separately as they have different functionalities and responsibilities (e.g. pokeball catches pokemons, pokefruit changes affection rate). Alternatively, these subclasses could be merged into a single subclass. However, doing so would have populated the objects with unrelated attributes and functionalities where not only would it affect the maintainability of code but it could also affect other classes that are related to it. Thus, this alternative was discarded. The final design is aligned with the Single Responsibility Principle as all extensions of the class focus solely on their unique attributes and behaviour.

Pokeball---<<captures>>--->Pokemon. Pokeball has an association with Pokemon. In the game, pokeball is an item required to capture pokemons, thus the final design implements a strong relationship between them.

FeedAction extends Action, CaptureAction extends Action, SummonAction extends Action. FeedAction, CaptureAction and SummonAction are concrete classes that extend the abstract Action class as they share a set of common attributes and methods with one another. Extending them from the same abstract class avoids repetition which supports the DRY principle. These classes have different functionalities (e.g. FeedAction feeds pokemon, CaptureAction catches pokemon, SummonAction summons pokemon). Alternatively, all these actions could have been created from a general action class. However, this alternative was rejected as doing so would populate the actions with unrelated functionalities as they are only implemented for certain actions (e.g. FeedAction can only feed pokemon). The final design involves separating the actions into different classes which are aligned with the Single Responsibility Principle, as they could have their own unique set of relatable attributes and methods that only performs the actions they are capable of doing.

FeedAction---<<feed>>--->Pokefruit. FeedAction has an association with Pokefruit. FeedAction is an action allowing the players to feed the pokemons with pokefruits. This action can only be executed if the player has a pokefruit in his/her inventory. Pokefruit and FeedAction have a strong connection between them, thus it is logical to implement an association relationship over dependency.

FeedAction---<<target>>--->Pokemon, CaptureAction---<<target>>--->Pokemon, SummonAction---<<target>>--->Pokemon. FeedAction, CaptureAction and SummonAction have associations with Pokemon. In the game, players execute FeedAction to a targeted pokemon to feed them, in hopes to increase their affection rate so they would have a higher probability of catching the pokemons. To capture a pokemon, players would execute CaptureAction that targets a pokemon. To summon a pokemon next to the player (trainer), players would execute a SummonAction that targets a pokemon. These actions are closely connected to the Pokemon, thus association relationships are implemented over dependencies.

Charmeleon---<<use>>--->AffectionLevel, Charizard---<<use>>--->AffectionLevel. Charmeleon and Charizard have a dependency on AffectionLevel enumeration. AffectionLevel is created as an enum class as each affection level has a constant range of affection points and should not be modified throughout the game. The enum class has dependency relationships with all the pokemons. The reason is that the pokemons' behaviour towards the player or trainer depends on its affection level. The way pokemons treat its trainer changes with its affection level towards the trainer.

Fire extends Item. Fire is a concrete class extending the abstract Item class, a base class that contains a set of common attributes and methods for all item objects. This class is extended separately as it has different functionalities and responsibilities as compared to the

other items in the game. Fire instances are created and added onto the Charizard's surroundings when it equips its special weapon, Fire Spin. The displayChar of the fire instance 'v' should overwrite the ground's displayChar for that particular turn. An alternative way to implement this is to create a new method (setDisplayChar) in each of the ground concrete subclasses. However, this is not possible as the method was already existent and protected inside the engine package. The second alternative is to create the fire instances as a special weapon using the SpecialWeapon class. This design was also aborted. Although the fire instances will have damage to some of the Pokemon, it still does not make sense to have it as a weapon when it cannot be equipped. Thus, the final design is to create a new concrete class and make the fire instances as items.

EvolveBehaviour---<<implement>>--->Behaviour. EvolveBehaviour is a concrete class that implements the Behaviour interface, in deciding whether or not to return an evolveAction in the current turn. An alternative was to skip this class and directly implement the EvolveAction class. However, this implementation was discarded as the actions for the actor to perform in the current turn is dependent on the priority of the behaviours. Actions relevant to a prioritised behaviour would be executed in that turn, given that the action is allowable. Thus, the final design included an EvolveBehaviour class which is responsible for checking the eligibility of performing that action in each turn before execution.

EvolveAction extends Action. EvolveAction is a concrete class that extends the abstract Action class since they share a set of common attributes and methods. This avoids repetition which supports the DRY principle. The EvolveAction class has a different functionality than the other action classes (e.g. EvolveAction evolves pokemon only). It is not recommended to create the action instance solely from the action class as doing so would populate the action with unrelated functionalities (e.g. having methods or attributes related to feeding, releasing, catching, etc.). Thus, the final design is implemented by creating a new class just for evolving action, which further aligns with the Single Responsibility Principle as it could have its own unique, relatable attributes and methods that only focuses on the actions it is capable of performing.

EvolveAction---<<target>>--->Pokemon. EvolveAction has an association with Pokemon. In the Pokemon game, this action is executed automatically when the Pokemon, specifically Charmander and Charmeleon, have survived for at least 20 turns, under one condition where no players or pokemon exist in their surroundings. Another way to evolve them is that players can manually perform this action when Charmander's and Charmeleon's affection points reach 100AP. This action is closely related to the Pokemon, thus an association relationship is implemented over dependency.

2.5 Day and Night

Charmander extends Pokemon, Squirtle extends Pokemon, Bulbasaur extends Pokemon and **Pokemon extends Actor.** Pokemon abstract class inherits Actor abstract class that has the necessary methods to generate an actor. Charmander, Squirtle and Bulbasaur concrete classes inherit a Pokemon abstract class that has the necessary method to generate a pokemon. This is aligned with the open-closed principle because the abstract classes don't allow modification, but are open for extension.

Lava extends Ground, Puddle extends Ground, Dirt extends Ground, Hay extends Ground. Lava, Puddle, Dirt and Hay concrete classes inherit the Ground abstract class that has the necessary methods to generate a type of terrain.

Puddle---<<create>>--->Dirt. From the game map, replace Puddle with Dirt. And, it is not needed for a strong relationship, a dependency relationship is enough. However, it is also an alternative way to create an association relationship between Puddle and Dirt concrete classes. Puddle will need to use the Dirt class as an attribute. After checking whether the actor is on the location through if-statement, use a random number generator to keep the 10% chance and change the Puddle to Dirt inside the statement. In order to prevent high coupling, it is better to use a dependency relationship between Puddle and Dirt concrete classes.

Tree extends SpawningGround. Tree is a dynamic ground. And, it inherits SpawningGround that has the necessary methods to form a dynamic ground.

Tree---<<create>>--->Hay and **Tree---<<create>>--->Candy.** From the game map, replace Tree with Hay or replace Tree with Candy. Both of them are not required for a strong relationship, a dependency relationship is enough.

Lava---<<use>>--->Utils, Puddle---<<use>>--->Utils, Tree---<<use>>--->Utils. These 3 terrains will expand based on a possibility, hence they need to use Utils class to check the possibility percentage. And, they only require reading the Utils class, so that a dependency relationship is enough.

Lava---<<use>>--->CapabilityOfExpand, Puddle---<<use>>--->CapabilityOfExpand, Tree---<<use>>--->CapabilityOfExpand. These 3 terrains can be expanded during the day or night. And, they need to use the CapabilityOfExpand enumeration to determine whether the location it expands is available to expand.

Floor---<<use>>--->CapabilityOfExpand, Wall---<<use>>--->CapabilityOfExpand. These 2 terrains are not expandable, hence adding a capability of not expandable is a better way to check whether this location can be expandable. An alternative way of doing this is use an “instanceof” method to check the location is not Floor and Wall. However, “instanceof” is a code smell and is a bad practice of using it. It is better to avoid using this method to perform a good design.

Charmander---<<implement>>--->TimePerception,
Squirtle---<<implement>>--->TimePerception,
Bulbasaur---<<implement>>--->TimePerception,
Lava---<<implement>>--->TimePerception,
Puddle---<<implement>>--->TimePerception, Tree---<<implement>>--->TimePerception. These concrete classes have different behaviours during day and night, and they perform them through TimePerception interface.

TimePerceptionManager---<<use>>--->TimePerception. TimePerceptionManager will calculate the turns and shift day or night automatically every 5 turns. And, it uses the TimePerception interface to control all the affected terrain and pokemon classes. Hence, enumeration TimePeriod is not needed anymore.

Player---<<use>>--->TimePerceptionManager. Player will run every turn. Hence, it is the best way to use the Player class to control the TimePerceptionManager class to count the turn and change day or night every 5 turns.

2.6 NurseJoy

NurseJoy extends Actor. NurseJoy extends the actor abstract class and its Playturn is DoNothingAction because it does not move. It will add TradeFirePokefruitAction, TradeWaterPokefruitAction, TradeGrassPokefruitAction, TradeCharmanderAction to the player when it is near the player to allow the player to trade with her.

NurseJoy---<<use>>--->Status. NurseJoy has a Status.IMMUNE capability to prevent it from being attacked.

TradeFirePokefruitAction extends Action, TradeWaterPokefruitAction extends Action, TradeGrassPokefruitAction extends Action, TradeCharmanderAction extends Action. TradeAction extends the Action abstract class. This is to allow the subclasses which inherit the Actor class to be able to access this class through ActionList in the Actor class that they inherited from. On a successful trade, the targeted item to trade will be instantiated and picked up by the Player immediately. On a failed trade, an error message will be printed.

TradeFirePokefruitAction---<<has>>--->Item(Pokefruit),
TradeWaterPokefruitAction---<<has>>--->Item(Pokefruit),
TradeGrassPokefruitAction---<<has>>--->Item(Pokefruit),
TradeCharmanderAction---<<has>>--->Item(Pokeball). All of the TradeActions has an attribute of Item as it stores the merchandise into the Player's inventory on a successful trade. The TradeAction and Item have an association relationship. In TradeCharmanderAction's special case a pokeball containing Charmander is added into the Player's inventory

TradeFirePokefruitAction---<<remove>>--->Candy,
TradeWaterPokefruitAction---<<remove>>--->Candy,
TradeGrassPokefruitAction---<<remove>>--->Candy,
TradeCharmanderAction---<<remove>>--->Candy. All of the TradeActions will remove Candy from the Player's inventory on a successful trade. The amount of Candy removed depends on the Object involved in the TradeAction through the implementation of the Tradable interface.

The implementation above adheres to the Single Responsibility Principle and Open-closed Principle of the SOLID principles. This is because we can avoid using multiple if-else statements for the TradeAction to determine the price of each item and the result of the trade. The price of each item can be determined by using a tradable interface implemented by the corresponding merchandise which is the Pokefruits and the Charmander. We are able to check the respective price of each merchandise without having to change any attributes in the Item abstract class. Single Responsibility Principle can be fulfilled because the details for trading is contained within the NurseJoy and all the TradeAction class.

Pokefruit---<<implements>>--->Tradable, Pokefruit implements the Tradable interface and implements its tradedWith method. Pokefruit's tradedWith() method will add the pokefruit directly to the Player's inventory on a successful trade.

Bulbasaur---<<implements>>--->Tradable, Squirtle---<<implements>>--->Tradable, Charmander---<<implements>>--->Tradable, Bulbasaur, Squirtle, Charmander all implements the Tradable interface and implement its tradedWith method. Their tradedWith() method will instantiate a new PokeEgg with their corresponding Pokemon inside and add the newly created PokeEgg to the Player's inventory.

TradeAction extends Action. TradeAction extends the Action abstract class. This is to allow the subclasses which inherit the Actor class to be able to access this class through ActionList in the Actor class that they inherited from. On a successful trade, the targeted item to trade will run its tradedWith() method to add the results of the trade to the Players inventory. On a failed trade, an error message will be printed.

TradeAction---<<has>>--->Actor and **TradeAction---<<has>>--->Tradable.** TradeAction has an attribute of Actor as the Player to be traded to and an attribute of any objects that implement the Tradable interface.

TradeAction---<<use>>--->Status, TradeAction---<<remove>>--->Candy. TradeAction uses Status.CURRENCY to determine how many candies are in the Player's inventory. will remove Candy from the Player's inventory on a successful trade. The amount of Candy removed depends on the Item involved in the TradeAction.

The implementation above adheres to the Single Responsibility Principle and Open-closed Principle of the SOLID principles. This is because we can avoid using multiple if-else statements for the TradeAction to determine the price of each item and the result of the trade. The contents of the trade and the price of each item can be determined by the arguments entered when creating a new action to be added inNurseJoy;s action list in her allowableActions() method. We are able to determine the contents of the trade which only includes objects that implement the tradable interface and their respective price without having to change any attributes in the Item abstract class. Single Responsibility Principle can be fulfilled because the price for trading is contained within the NurseJoy and TradeAction class.

An alternative for the implementation of NurseJoy is adding an attribute of price for the Item abstract class so any of its subclasses will have a price for trading. However, this implementation method is less desirable as it violates the Single Responsibility principle and as the attribute only used for trading is now inherited by all the item's subclasses and some of the subclasses of Item are not tradable. Besides that, this implementation requires modification on Item abstract class for it to work and Open-closed principle does not allow that.

2.7 Pokemon Egg and Incubator

PokemonEgg extends Item. PokemonEgg concrete class inherits Item abstract class. Pokemon will be instantiated upon a successful TradeAction between the Player and NurseJoy and added to the Player's inventory. PokemonEgg has an attribute of hatchTime to store the amount of time it is needed to hatch.

PokemonEgg---<<use>>--->BasePokemon. BasePokemon is an Enum class which stores all the base pokemon and their corresponding hatchTime inside their respective Enum. When instantiating a PokemonEgg object, you need to pass in a BasePokemon Enum to ensure only base Pokemons can be born from the PokemonEgg. The constructor of PokemonEgg will also store the Pokemon and hatchTime inside the object itself so the PokemonEgg knows what type of PokemonEgg it is and its corresponding hatchTime.

BasePokemon---<<has>>--->BasePokemon.

PokemonEgg---<<has>>--->Pokemon. PokemonEgg object has an attribute of Pokemon to store which Pokemon it will be spawning once it hatches.

PokemonEgg---<<hatches>>--->Pokemon. PokemonEgg's tick() method has been overridden so it adds one to its hatchTimer counter if it is currently on a Incubator ground. When a PokemonEgg reaches the required number of turns to hatch on an Incubator ground, it will spawn a new Pokemon of its corresponding type on top or around its current location while also removing itself.

Incubator extends Ground. Incubator concrete class inheritsGround abstract class and inherits Ground abstract class's attributes and method.

Incubator---<<use>>--->Status. Incubator has a Status.HATCHING_GROUND capability to allow other classes to know it is an Incubator class object. The PokemonEgg will undergo its hatching process if it is on top of a ground with Status.HATCHING_GROUND capability.

2.8 New Trainer

Trainer extends Actor. Trainer abstract class inherits the actor abstract class and all of its methods and attributes.

Trainer---<<use>>--->Status. NurseJoy has a Status.IMMUNE capability to prevent it from being attacked.

Trainer---<<use>>--->Character. Incubator has a Character.TRAINER capability to allow other classes to know it is a Trainer class object. This allows us to add an action to view the Trainer's stats from the Player class's playTurn method.

Trainer---<<has>>--->Behaviour. Trainer has a TreeMap of Integer to Behaviour. The trainer utilises this map to decide which action to automatically perform. Each behaviour will be able to create new actions if they meet the requirements. The trainer will loop through the map of behaviours until an action is returned from one of the behaviours No behaviours are added into Trainer abstract class's map as each different trainer may have different priorities and set of actions, thus their implementation may differ..

TrainerCatchBehaviour---<<implements>>--->Behaviour,
TrainerCatchBehaviour---<<creates>>--->CaptureAction. TrainerCatchBehaviour implements the Behaviour Interface and its getAction method. TrainerCatchBehaviour will check if the current actor's (Trainer) surroundings for pokemons with an affection of >75, if true it will return a newly created CaptureAction of that pokemon else it will return null.

TrainerCatchBehaviour---<<use>>--->Character,
TrainerCatchBehaviour---<<use>>--->Pokemon,
TrainerCatchBehaviour---<<use>>--->AffectionManager,
TrainerCatchBehaviour---<<use>>--->AffectionLevel. TrainerCatchBehaviour uses Character to determine if the actor at a certain location is a Pokemon, then it uses AffectionManager instance to check if the Pokemon's affection level towards the trainer, if it reaches a specific fixed level set within AffectionLevel class, TrainerCatchBehaviour will return the new CaptureAction.

TrainerFeedBehaviour---<<implements>>--->Behaviour,
TrainerFeedBehaviour---<<creates>>--->FeedAction. TrainerFeedBehaviour implements the Behaviour Interface and its getAction method. TrainerFeedBehaviour will check if the current actor's (Trainer) inventory first and his surroundings for pokemons, if both checks return true it will check all of the Trainer's Pokefruits and if any of them has a matching Element with the current Pokemon. If any of the Pokefruits has a matching element it will return a newly created FeedAction of that pokemon and the pokefruit else it will return a newly created FeedAction with a random Pokefruit from the trainer's inventory. If it does not fulfil the first check it will return null.

TrainerFeedBehaviour---<<use>>--->Character,
TrainerFeedBehaviour---<<use>>--->Pokemon,
TrainerFeedBehaviour---<<use>>--->Status,
TrainerFeedBehaviour---<<use>>--->Pokefruit,
TrainerFeedBehaviour---<<use>>--->Element,
TrainerFeedBehaviour---<<use>>--->ElementsHelper,

TrainerFeedBehaviour---<<use>>--->Utils. TrainerFeedBehaviour uses Status to check if the current actor(Trainer) has a Pokefruit in his inventory, then it usesCharacter to determine if the actor at a certain location is a Pokemon. After that, it uses ElementsHelper to check if any of the Pokefruits has a similar Element to the Pokemon. TrainerCatchBehaviour will return the new CaptureAction with a Pokefruit with a similar Element to the Pokemon or else it will use a random Pokefruit from the Trainer's inventory for the action. If either there is no Pokefruits in the trainer's inventory or there are no Pokemon's in the vicinity of the Trainer it will return null.

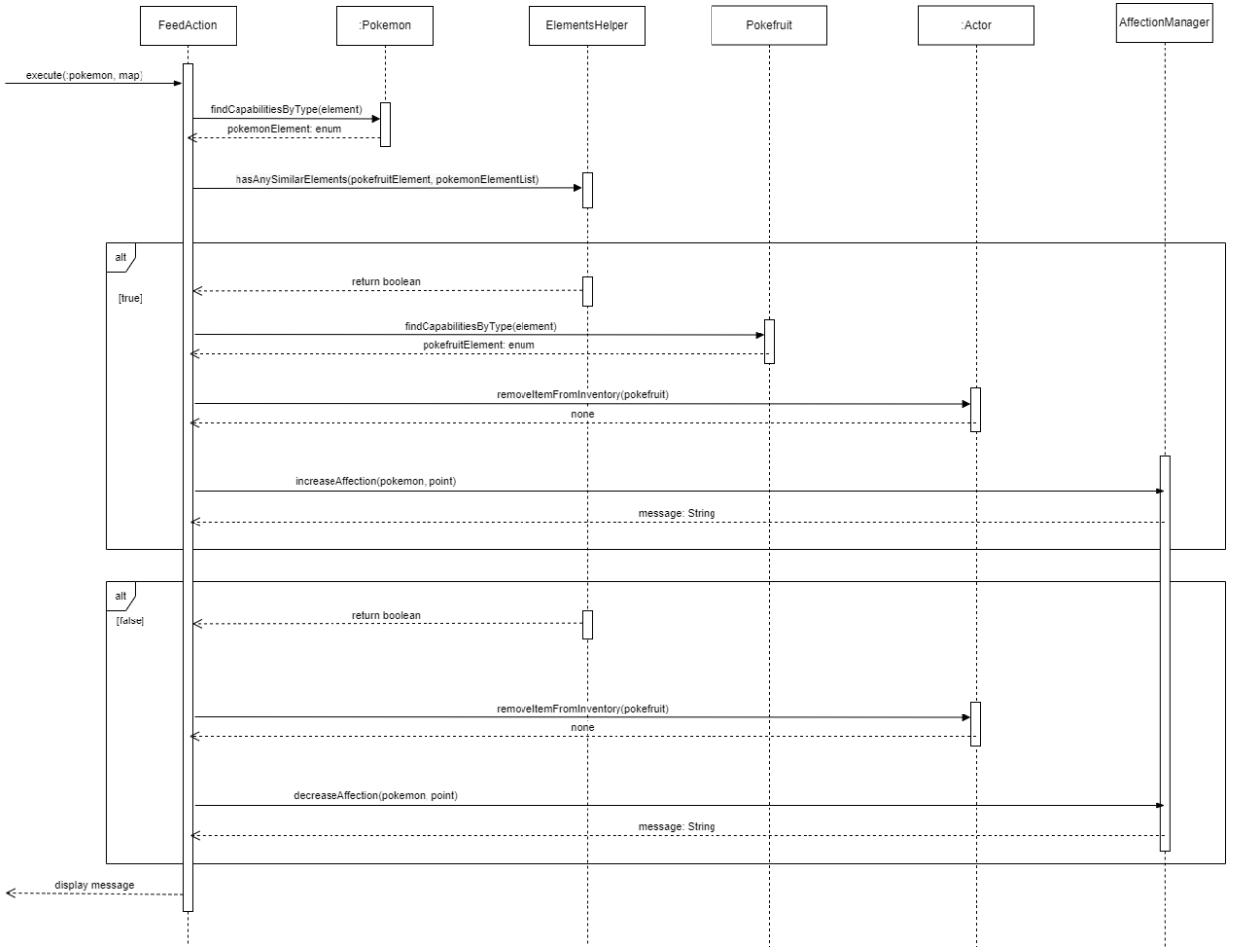
TrainerPickupBehaviour---<<implements>>--->Behaviour,
TrainerPickupBehaviour---<<uses>>--->Item,
TrainerPickupBehaviour---<<uses>>--->PickupItemAction. TrainerPickupBehaviour implements the Behaviour Interface and its getAction method. TrainerFeedBehaviour will check the current actor's (Trainer) location for any items that are able to be picked up. It will return A PickupAction of the first item that can be picked up or else it will return null.

TrainerGoh extends Trainer. TrainerGoh concrete class inherits the Trainer abstract class and all of its methods and attributes. TrainerGoh is an implementation of Trainer abstract class.

TrainerGoh---<<use>>--->TrainerCaptureBehaviour,
TrainerGoh---<<use>>--->TrainerFeedBehaviour,
TrainerGoh---<<use>>--->TrainerPickupBehaviour,
TrainerGoh---<<use>>--->WanderBehaviour, TrainerGoh has TrainerCaptureBehaviour, TrainerFeedBehaviour, TrainerPickupBehaviour and WanderBehaviour in smallest to largest order in its TreeMap of Integers to Behaviours. TrainerGoh's playTurn() method will loop through this map in order and execute the first action returned from the map of behaviours. It will try to find a pokemon to capture, else try to feed a pokemon, else try to pick up an item from the ground, if it cannot do any of the above it will move in a random direction. If it still cannot move due to being surrounded on all sides it will return a new DoNothingAction.

3 Sequence Diagram

FeedPokemon



Hatch Pokemon from Pokemon Egg