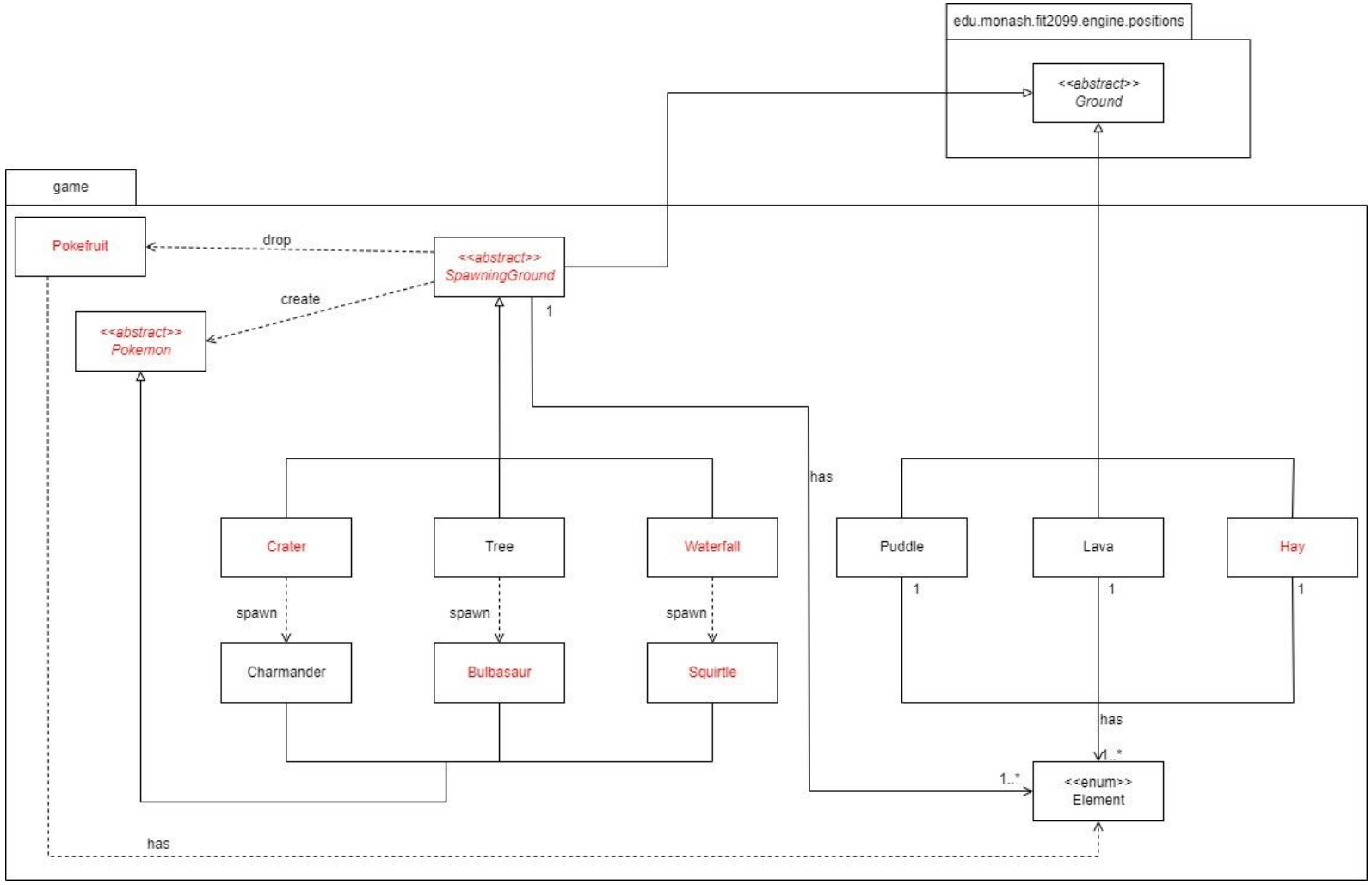
Assignment 1
FIT2099
Object Oriented Design and Implementation

Team members:

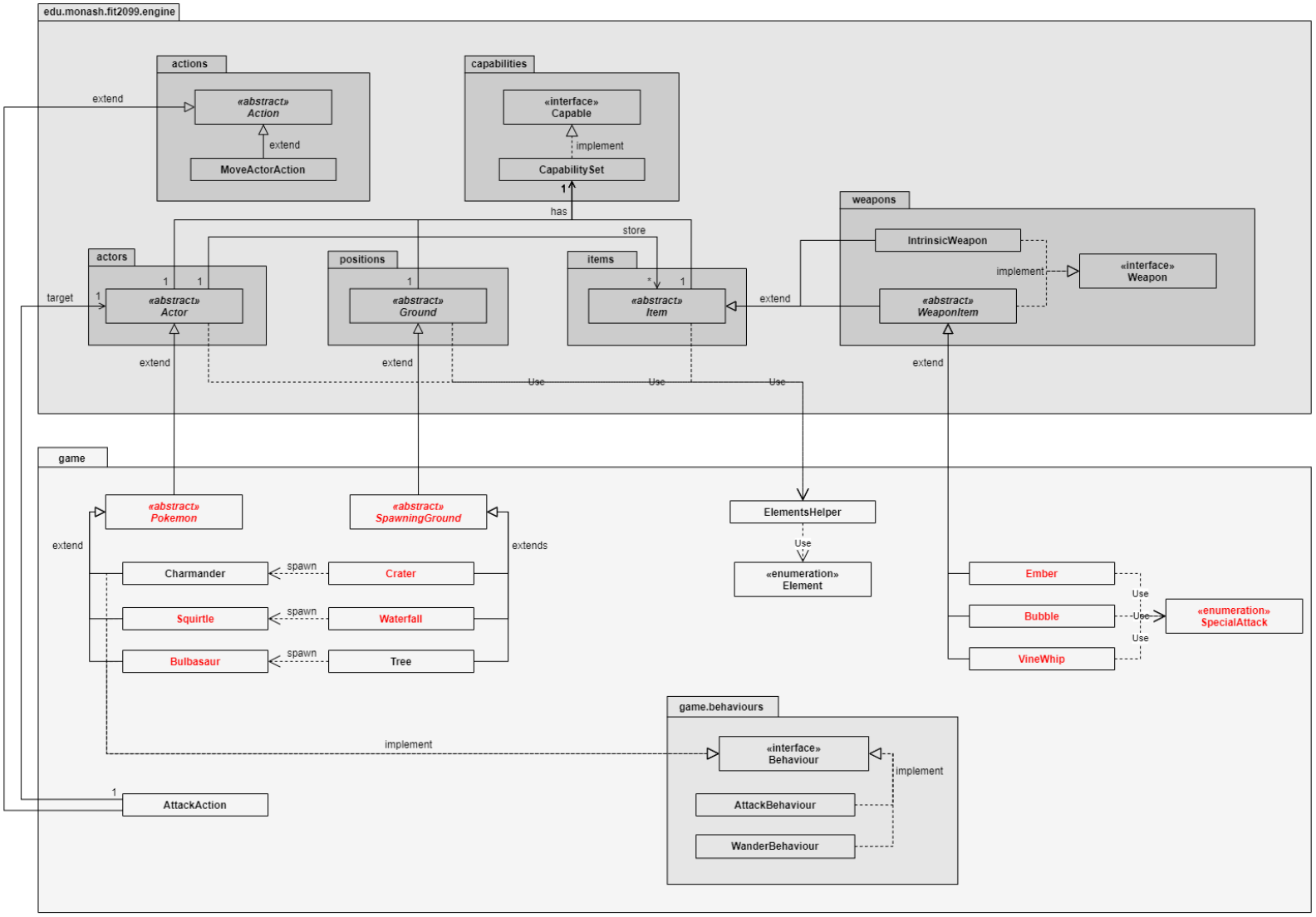
1. Fong Zhiwei (32686072)
2. Soh Meng Jienq (30531608)
3. Leong Xin Yun (32835477)

1 UML Diagrams

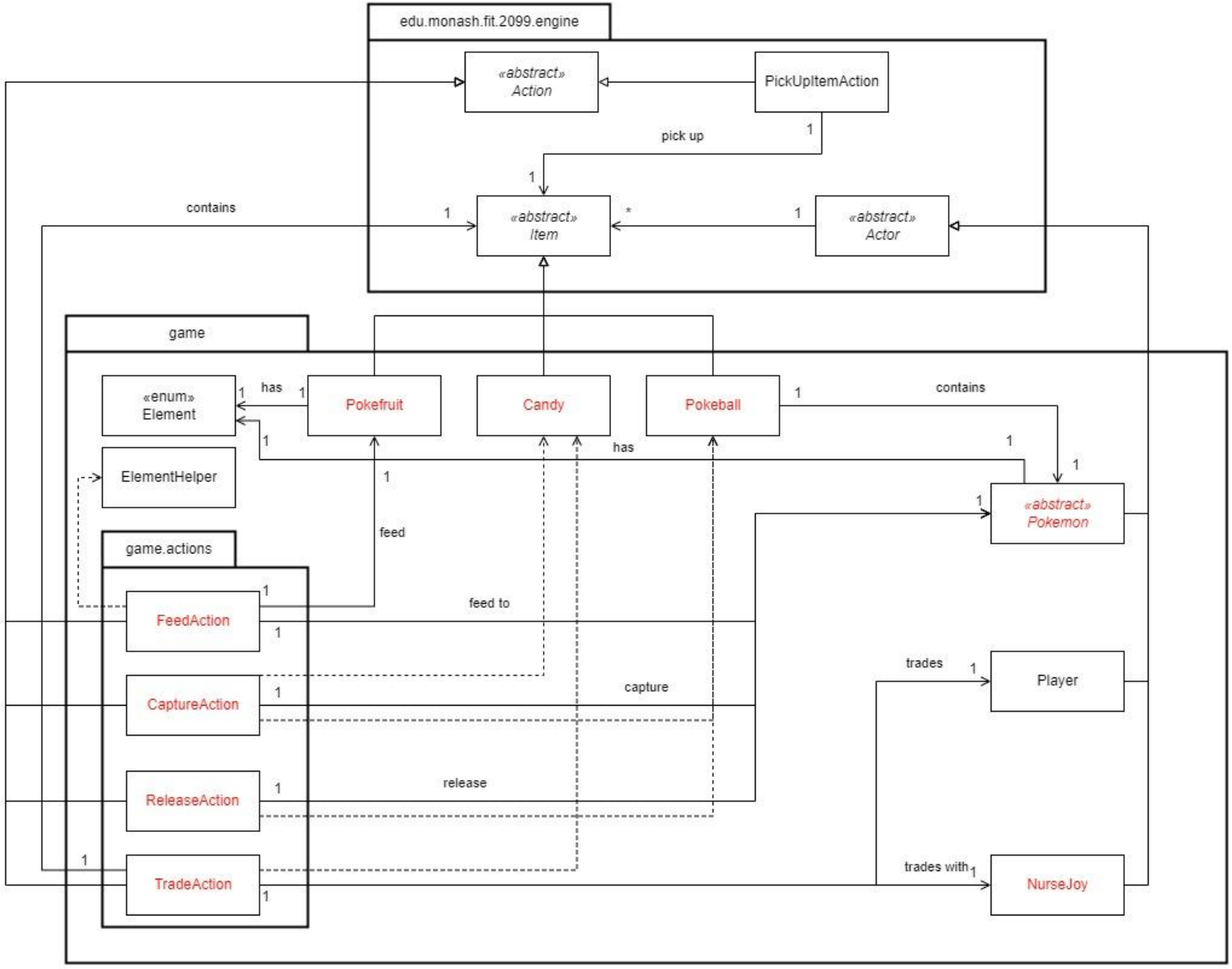
1.1 Environment



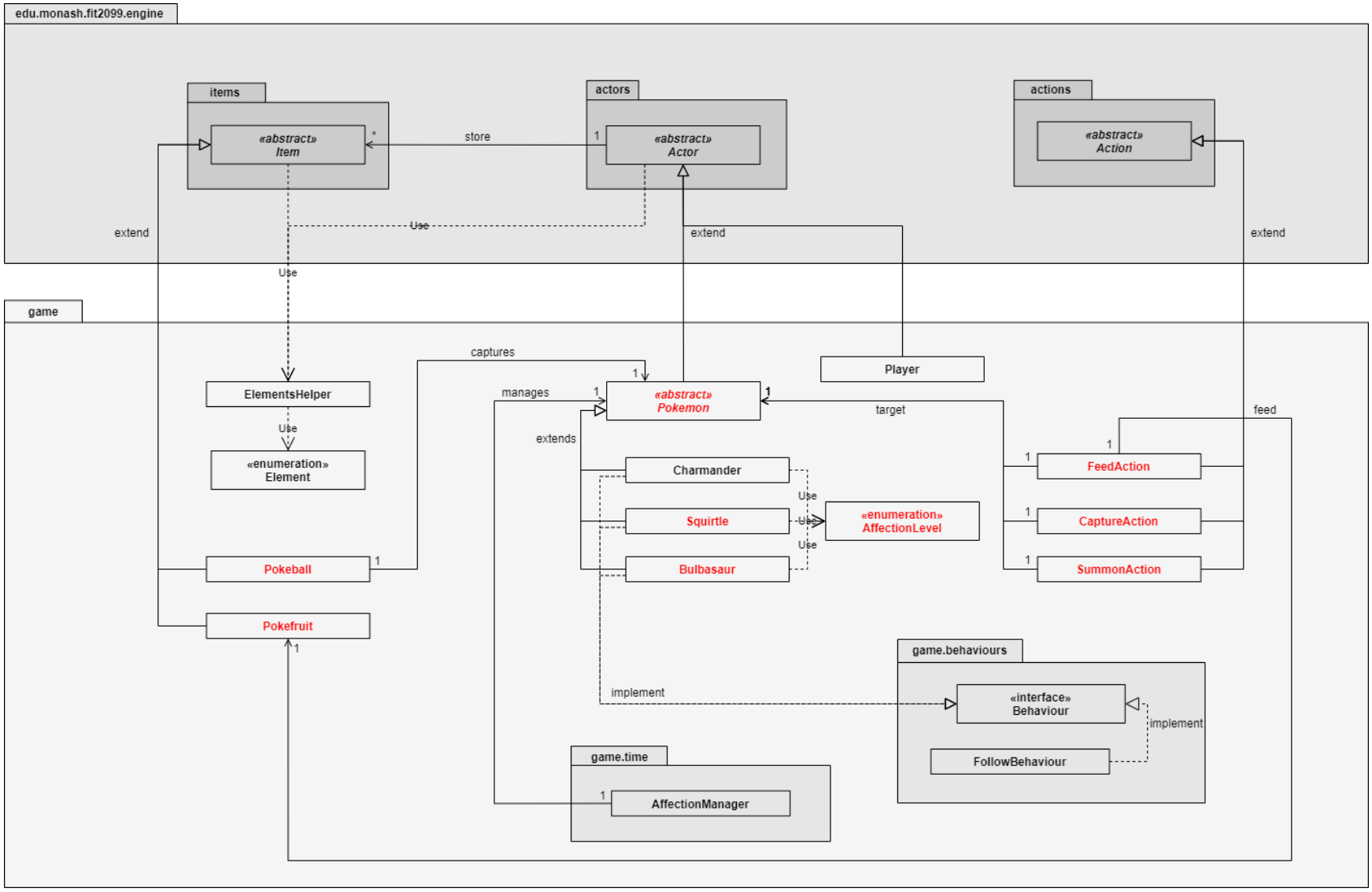
1.2 Pokemons


edu.monash.fit2099.engine
actions
«abstract»
Action
extend
MoveActorAction
capabilities
«interface»
Capable
implement
CapabilitySet
has
store
weapons
IntrinsicWeapon
implement
«interface»
Weapon
«abstract»
WeaponItem
actors
«abstract»
Actor
target
positions
«abstract»
Ground
items
«abstract»
Item
extend
Use
game
«abstract»
Pokemon
«abstract»
SpawningGround
extends
Charmander
Squirtle
Bulbasaur
spawn
Crater
Waterfall
Tree
ElementsHelper
«enumeration»
Element
Ember
Bubble
VineWhip
«enumeration»
SpecialAttack
game.behaviours
«interface»
Behaviour
AttackBehaviour
WanderBehaviour
AttackAction

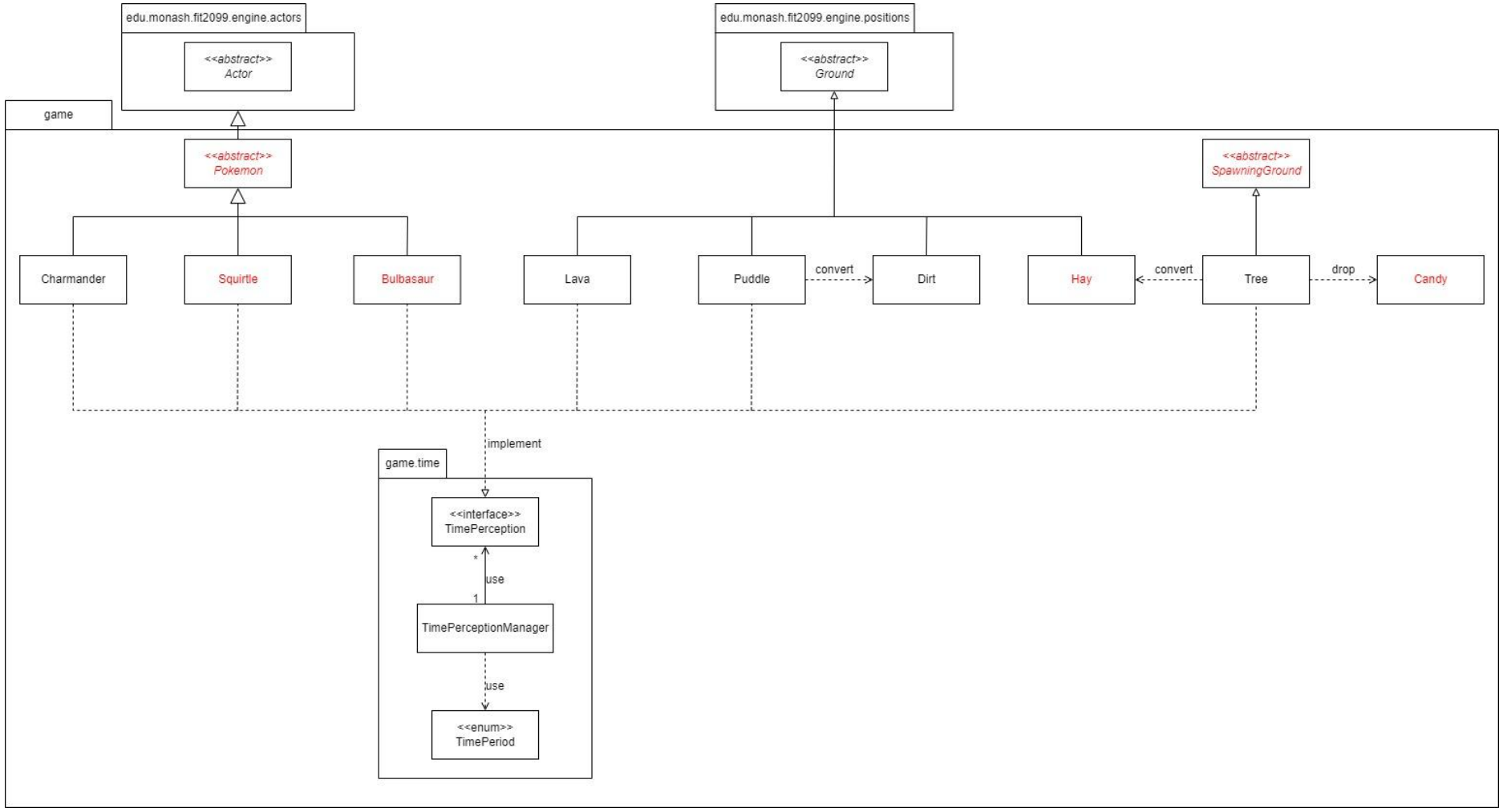
1.3 Item


edu.monash.fit.2099.engine
«abstract» Action
PickUpItemAction
pick up
1
1
contains
1
«abstract» Item
*
1
«abstract» Actor
game
«enum» Element
1
has
1
Pokefruit
Candy
Pokeball
1
contains
1
has
1
1
ElementHelper
1
1
«abstract» Pokemon
feed
game.actions
1
feed to
FeedAction
1
trades
1
Player
CaptureAction
1
capture
ReleaseAction
1
release
1
TradeAction
1
trades with
1
NurseJoy

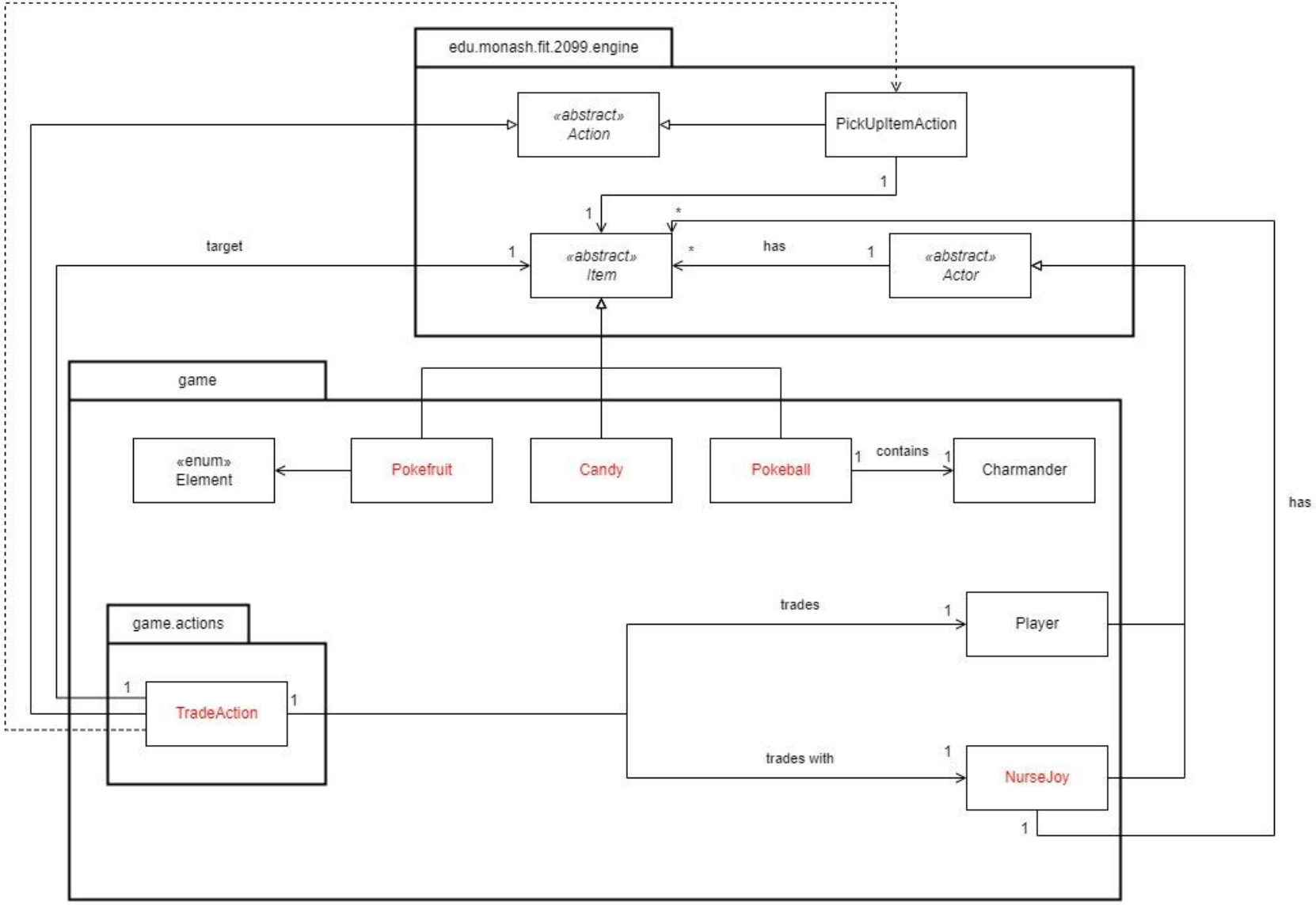
1.4 Interactions



1.5 Day and Night



1.6 Nurse Joy


edu.monash.fit.2099.engine
«abstract»
Action
PickUpItemAction
1
1
*
target
1
«abstract»
Item
*
has
1
«abstract»
Actor
game
«enum»
Element
Pokefruit
Candy
Pokeball
1
contains
1
Charmander
has
trades
1
Player
game.actions
1
TradeAction
1
trades with
1
NurseJoy
1

2 Design rationale

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---<<spawn>>--->Charmander, **Tree---<<spawn>>--->Bulbasaur** and **Waterfall---<<spawn>>--->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---<<drop>>--->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---<<drop>>--->Pokefruit and **Pokefruit---<<has>>--->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.

SpawningGround---<<has>>--->Element. The SpawningGround abstract class itself also has an element (Fire, Water, Grass). And, it needs to generate different element types of SpawningGround, thus it needs to have a stronger relationship, which is the association relationship.

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---<<has>>--->Element, Lava---<<has>>--->Element, Puddle---<<has>>--->Element. Hay, Lava and Puddle have three different types of element. And, they need to have a stronger relationship with Element class to add the capability of the element to each terrain. Hence, they use an association relationship with the Element class.

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---<<acts>>--->Behaviour, Squirtle---<<acts>>--->Behaviour, Bulbasaur---<<acts>>--->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---<<spawns>>--->Charmander, Waterfall---<<spawns>>--->Squirtle, Tree---<<spawns>>--->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.

Ember extends WeaponItem, Bubble extends WeaponItem, VineWhip extends WeaponItem. Ember, Bubble and VineWhip extended the abstract class WeaponItem because they share a common set of attributes and methods with one another. Extending them from the same abstract class avoids repetition which supports the DRY principle. Alternatively, all the weapons could have been created from a general weapon concrete class. This implementation was rejected as doing so would populate the weapons with unrelated behaviours as they are not capable of doing all special attacks (e.g. Ember can only be equipped and used by Charmander on Fire element ground). The final design involves splitting the weapons into different specific classes which are aligned with the Single Responsibility Principle, as they could have a set of relatable attributes and methods that only focuses on the attacks they are capable of.

Ember---<<use>>--->SpecialAttack, Bubble---<<use>>--->SpecialAttack, VineWhip---<<use>>--->SpecialAttack. Ember, Bubble and VineWhip have a dependency on SpecialAttack enumeration. SpecialAttack is created as an enum class because every special attack acts as a constant and should not be modified throughout the game. The enum class has dependency relationships with all the weapons, namely Ember, Bubble and VineWhip. The reason being is that the attributes and methods of a weapon change with the type of attack a pokemon is capable of performing.

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.

ReleaseAction---<<remove>>--->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. 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.

Pokefruit---<<has>>--->Element. Each Pokefruit has a single element as its attribute to determine which type of Pokefruit it is. There is a association relationship between them.

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.

TradeAction---<<remove>>--->Candy. TradeAction 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.

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---<<acts>>--->Behaviour, Squirtle---<<acts>>--->Behaviour, Bulbasaur---<<acts>>---> 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 catch 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---<<feeds>>--->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---<<targets>>--->Pokemon, CaptureAction---<<targets>>--->Pokemon, SummonAction---<<targets>>--->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.

AffectionManager---<<manages>>--->Pokemon. AffectionManager has an association with Pokemon. This is an existing class in the game package which mainly focuses on the affection rate of a pokemon towards a player (e.g. Increasing and decreasing the affection rate are performed by AffectionManager).

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---<<convert>>--->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---<<convert>>--->Hay and **Tree---<<drop>>--->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.

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 and **TimePerceptionManager---<<use>>--->TimePeriod.** TimePerceptionManager will add an object instance (time) to an ArrayList. And, use TimePeriod to shift between day and night. TimePerceptionManager has a weaker relationship with TimePeriod, which is the dependency relationship, because it only reads TimePeriod and it will not modify anything inside the enumeration TimePeriod. However, TimePerceptionManager needs to have a stronger relationship with TimePerception interface, which is the association relationship, because TimePerceptionManager needs to manage the day and night effects of the TimePerception interface.

2.6 NurseJoy

NurseJoy extends Actor. NurseJoy extends the actor abstract class and has an ArrayList of items which acts as the items she is tradable with. NurseJoy has a HashMap to determine the price of the items with Item as key and price as value.

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 be instantiated and picked up by the Player immediately. On a failed trade, an error message will be printed.

TradeAction---<<has>>--->NurseJoy and **TradeAction---<<has>>--->Player.** TradeAction has an attribute of NurseJoy as NurseJoy and Player are the only characters that are allowed to trade. Specifying the attribute as NurseJoy and Actor allows the class to differentiate between the inventory of the trader(NurseJoy) and the inventory of the Player.

TradeAction---<<has>>--->Item. TradeAction has an attribute of Item as it allows the user to choose an item from the trader's(NurseJoy) inventory as the trade target. The TradeAction and Item have an association relationship.

TradeAction---<<remove>>--->Candy. TradeAction 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 relationship between TradeAction and Candy is shown through the relationship between TradeAction and Item.

TradeAction---<<knows>>--->PickupAction. TradeAction knows about PickupAction because the Player will pick up the Item immediately after a successful trade.

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 HashMap in NurseJoy we are able to check on every type of item 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.

3 Sequence Diagram