Design rationale

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.

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 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---<<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).