Assignment 2: Design Documents

FIT2099: Object-Oriented Design and Implementation

Team: CL_Lab2Group6
Zhijun Chen
Minh Tuan Le
Ishrat Kaur

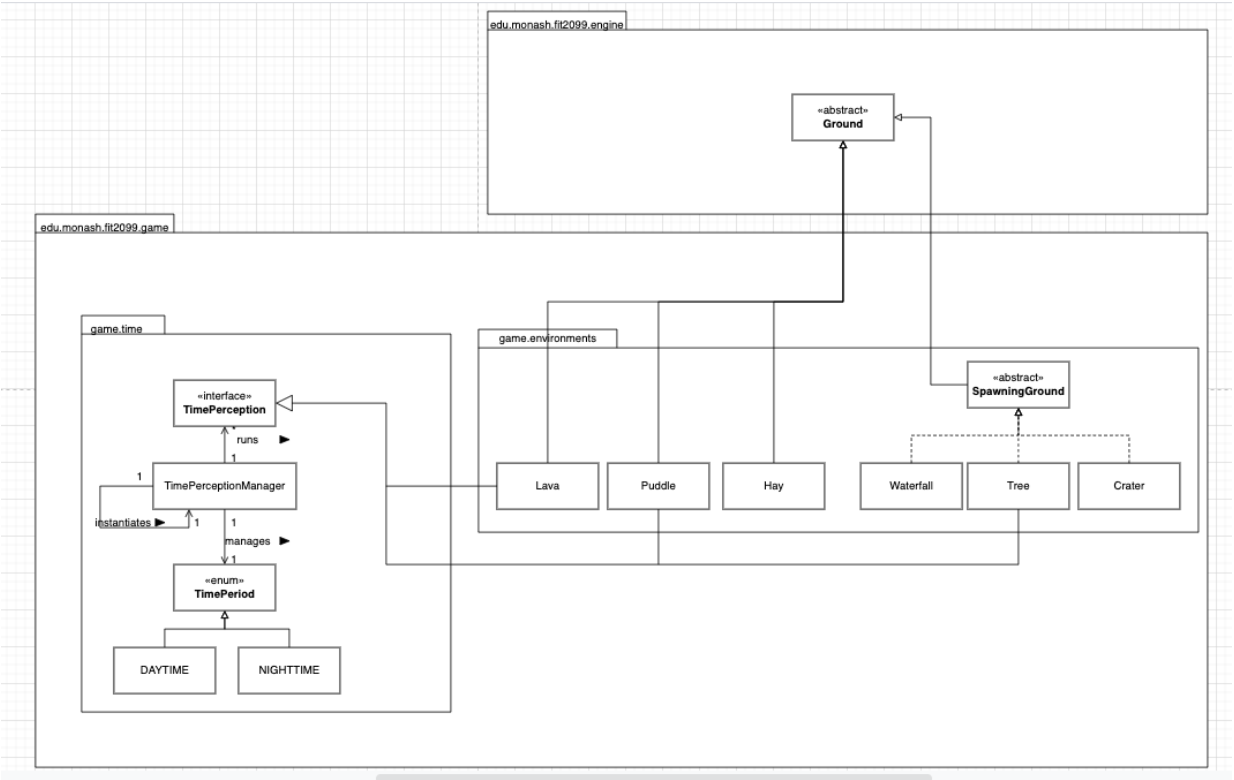
Table of contents

Design documentation

This document contains three UML (unified modelling language) diagrams for six requirements specified in the project. UML diagrams are followed by corresponding design rationales explaining major design decisions and the underlying principles.

REQ1 + REQ5:

Environment



Design rationale

The abstract class spawning ground is a subclass of ground and is a super class for 3 spawning ground(Crater, Waterfall, Tree), the abstract class spawning will have 2 abstract method which need to implement in the subclass, which is spawnPokemon() and dropPokefruit(), and a override method tick(Location location), this method pass the location of the spawning ground to this class, and call spawnPokemon() and dropPokefruit() to spawn pokemon and drop poke fruit every turn.

The subclass Waterfall and Crater only need to implement the abstract method from super class with their requirement (spawn rate and spawn requirement, drop rate), The subclass Tree also experience the day and night effect, so the subclass Tree will also implement the interface TimePerception, so the subclass tree will implement the day and night effect with the rate, and in order to achieve these effects if the rate reach the requirement, there will 2 method to complete the effect of day and night, which is dayTree(Location location) and nightTree(Location location).

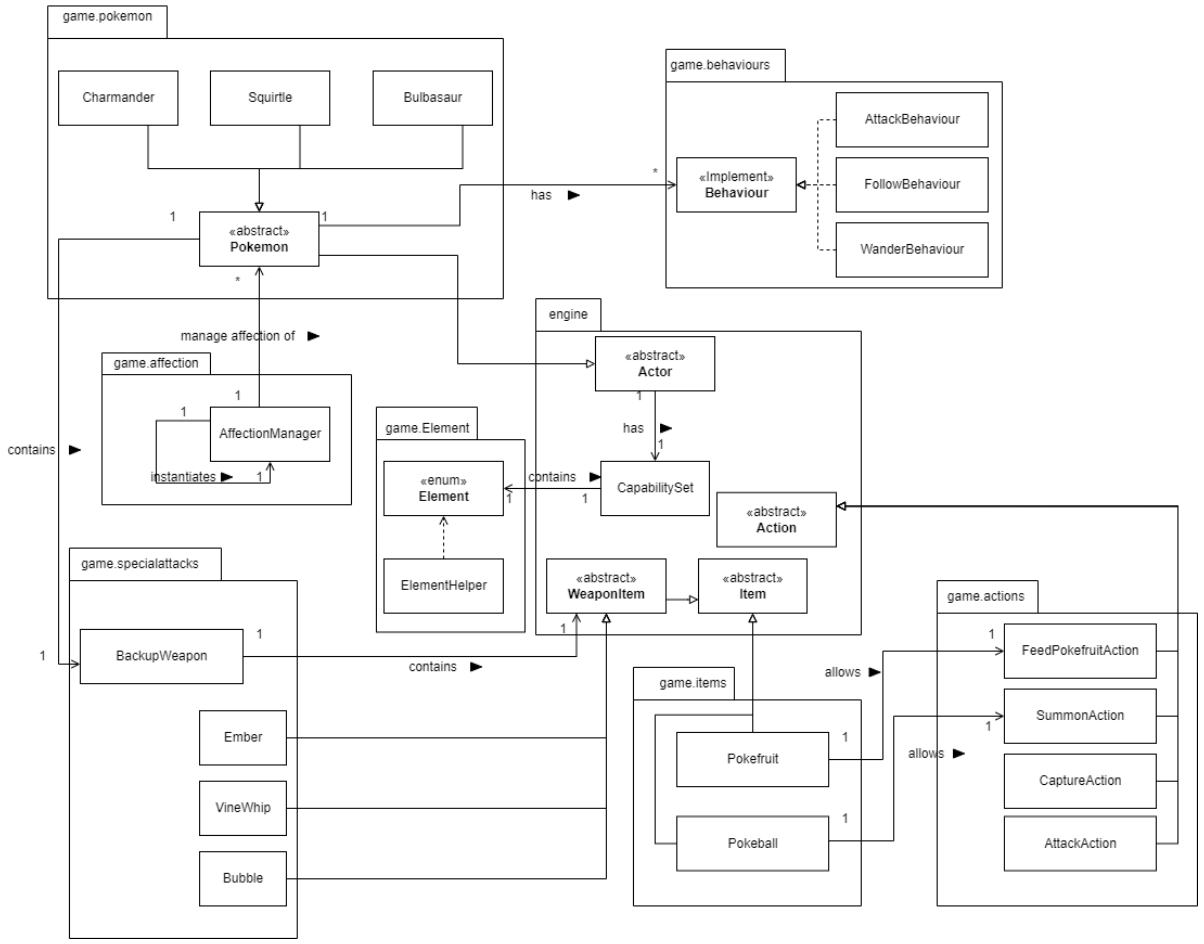
Lava, Puddle and Tree class will experience the day and night effect, so these class will implement the interface TimePerception, and override the method dayEffect() and nightEffect(), and these class will have their own method to achieve their effects with the param of location.

In the TimePerception interface, the dayEffect() and nightEffect() will be overridden with their own possibility in the class which experience the day and night effect, and the registerInstance() method will put these classes which experience the day and night effect to the time perception list.

In the TimePerceptionManager, the function run() will add the turn count and also recognize the day and night period for the class experience effect, and call dayEffect() or nightEffect() function for the class in the time perception list, and this method will run in playTurn to make sure it runs every turn. And it also has another function append(objInstance) to add object instances to the time perception list.

REQ2 + REQ4:

Pokemons and Interaction



Design rationale

The pokemon subclass is the superclass for the pokemon (Charmander, Squirtle, …) which is an abstract class of actor class and can contain additional attributes and methods unique for Pokemons. The Pokemon class maintains a list of Behaviours and its priority to manage the actions of the pokemons. All behaviours (FollowBehaviour, AttackBehaviour, WanderBehaviour) implement the Behaviour interface. This follows the Interface segregation principle by separating parallel behaviours with similar attributes and methods.

The Item subclass Pokeball can maintain a Pokemon class object to be summoned by SummonPokemonAction.execute(). The CaptureAction class object is allowed by Pokemon in its allowableAction() method if appropriate conditions are met.

The Special attack and intrinsic attack of the pokemon will be handled using the weapon system. As the game engine already implemented attack action using WeaponItem class, implementing special attacks (VineWhip, Ember, and Bubble) as a subclass of WeaponItem makes the design simpler and more compatible with the system. This is also in accordance with the Dependency Inversion Principle.

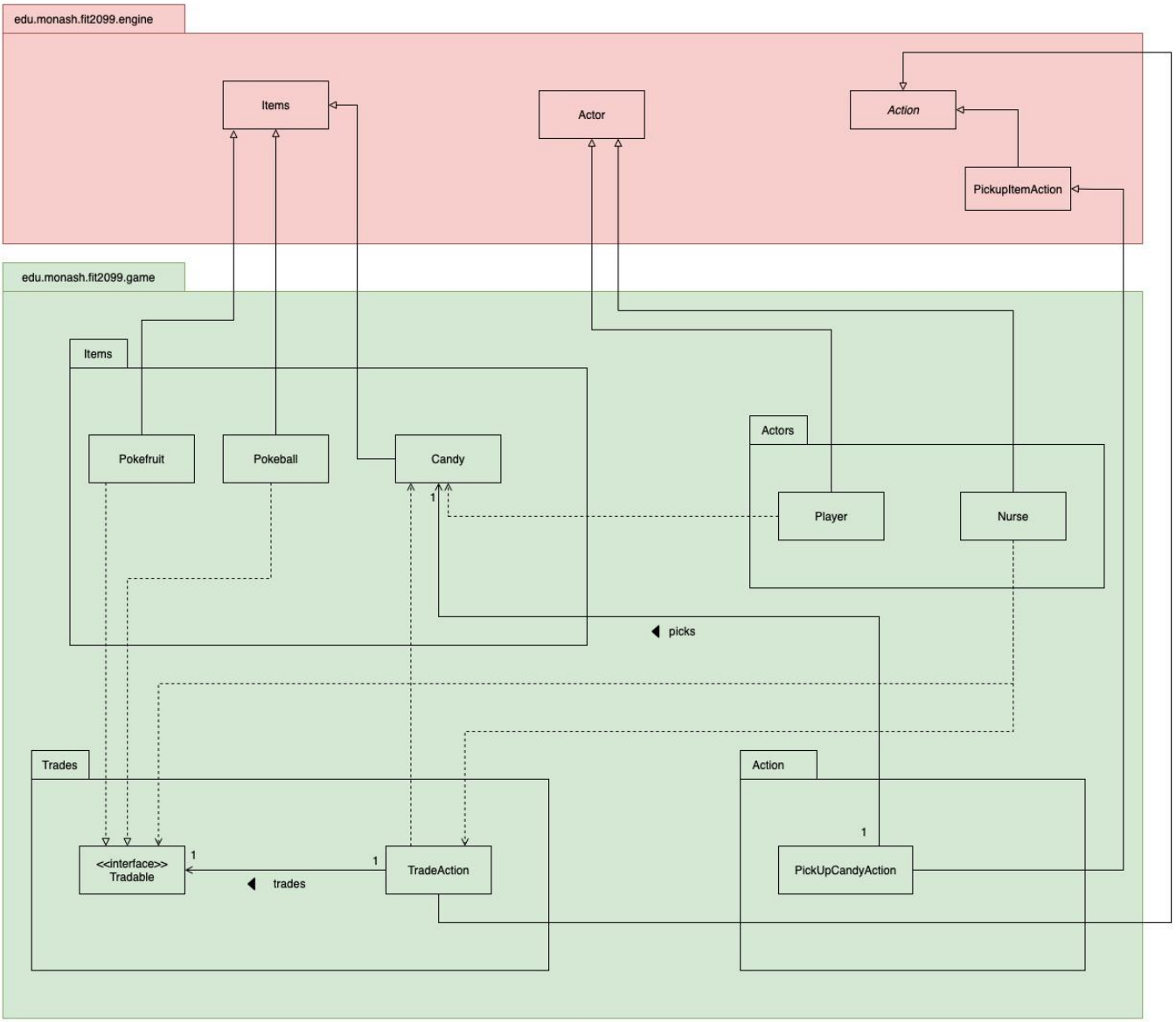
The AttackAction are added to the ActionList of the Player when they are next to the Pokemon, which is monitored and updated in the allowableActions() method. Each Pokemon maintains a behaviour list that is updated based on AffectionPoint by the AffectionManager method updatePokemonBehaviours.

The affection mechanism will also be handled by the AffectionManager class. Though this is a violation of the Single Responsibility principle, it would make the design simpler and more succinct if a single class with a single instance per game were to handle the affection mechanisms than having all Pokemon bearing an affection attribute or class and changing all its behaviours and statuses related to affection.

The Pokemon class will handle the special attack with the help of BackupWeapon storing one WeaponItem object class to ensure only one Special weapon is used for each Pokemon. This dependency keeps the game from being overloaded with WeaponItem and AttackAction objects.

REQ3 + REQ6:

Items



Design rationale

A key significant decision was made to add an interface Tradable which allows Pokefruit and Pokeball to implement methods in the Tradable interface. This decision allowed us to adhere to the dependency inversion principle which not only helps us add more abstraction to our code but also makes it easier to expand the game with addition of more tradable items in the future.
The PickUpCandyAction class was created to implement the action of picking up the candy by the player, again to adhere to the Single Responsibility Principle.

The TradeAction class was created to implement the process of trading between nurse joy and the player. An alternative approach could have been to implement the trading processes within the Player class. This alternative approach would have been a direct violation of Single responsibility principle as this would lead to many actions being performed by the Player class. Therefore, the TradeAction class allows the design to be more Object-Oriented. Also, should there be a bug in the code, with all classes adhering to the Single Responsibility Principle, it is easy to identify and fix errors. Overall, the association relationship between TradeAction and Tradable allows efficient design with abstraction to perform Trading action. The tradable items implement the Tradable interface. Therefore, instead of having direct multiple associations with tradable items, our approach follows dependency inversion principle and provides a Object-Oriented approach to implement the trading action of the game.