IERG3080 Project Part II

Design Report

Wong Keng Lam (1155116253), Wong Wan Ki (1155124843)

# 1. Functionalities and Use Cases

Our prototype focuses on the following four important elements in Pokemon Go: Navigation, Capture, Gym-battle and Manage Pokemon. These elements are implemented with some extra features incorporated into them. The details of each function and their use cases are listed below:

1. **Before game starts**
   i. A starting screen is shown prompting player to enter the username.
   ii. The program will be redirected to Navigation screen upon pressing "enter".
2. **Navigation**
   i. Players can walk the avatar around using arrow keys.
   ii. Different Pokemon is spawn randomly at regular intervals. Their location is hidden until player walks near them. Program will navigate to Capture mode once player collides with the Pokemon.
   iii. The Map is fixed in the background, with fixed "Battle Gym" and "Home" locations. Walking into those locations will cause the program to navigate to the corresponding mode.
   iv. Pokeballs are also spawned at random locations at regular intervals. Players can get 5 pokeballs once the avatar overlaps with the location of the Pokeball icon.
3. **Capture**
   i. A mini-game will immediately start, where players have to press the "enter" button at the right timing to make the gray circle and green circle overlap.
   ii. The performance is judged by the closeness of the two circles. The closer the two circles are, the higher the probability of successfully catching the Pokemon.
   iii. If successful, the Pokemon and 500 stardust will be added to player's home. Otherwise, players may play the mini-game again. There is also chance that the pokemon will escape, in which players will be redirected back to the navigation screen with no rewards.
4. **Gym-Battle**
   i. Before the battle, players can choose any one Pokemon from their collection to take part in the Gym battle. The battle will start once the player have chosen their pokemon.
   ii. The system will randomly choose a Pokemon to be the opponent Pokemon.
   iii. (Extra Feature) If player has collected enough Pokemons, there will be chance that an ultra-rare Pokemon will appear.
   iv. The gym battle is implemented as a turn-based battle. In each turn, players can choose from three possible moves that the Pokemon has. Each move has a limited number of usages.
   v. The game ends when any party has run out of HP. If players win, they will be rewarded with 1000 Stardust and the opponent Pokemon.
5. **Manage Pokémon/Collection**
   i. A collection screen will appear where users can scroll through their owned Pokemons, view stats and perform actions on them.
   ii. Available actions include Evolve, Power UP, Rename and Sell.
   iii. Evolve and Power UP raises stats of the Pokemon and both actions require certain stardust. Evolving a Pokemon may also change its look and its name. The HP of a pokemon will automatically recover to full after performing these two actions.
   iv. Players can also sell their Pokemon to gain additional Stardust.
6. **(Extra feature) Rarity of Pokemons**
   i. There are three type of rarity: common, rare, and ultra-rare.
   ii. Common and Rare pokemons will appear in the wild but rare pokemons only appear with a very low probability.
   iii. Ultra-rare pokemons only appear in gym-battles (see 4.iii).

## 2. Class designs

We designed various classes to facilitate the logic and data flow between the game. We classify the classes in two types: game-scope classes (classes that are used throughout the game) and view-scope classes (classes that are more attached to a particular view). We will discuss each of them below.

For game-scope classes, we mainly have the following three classes: PokemonType, Pokemon and Player. They are more focused in storing the data for retrieval in other views than the logic.

The PokemonType class models each different "species" of pokemon present in our game. It is used to to store the different characteristics of each type of Pokemon for convenience when instantiating a Pokemon. It therefore only has various fields but without any methods. Objects of the PokemonType class are instantiated at the beginning of the game using data stored in a csv file and stored in a set for retrieval, and new objects will not be instantiated again any point during the game. This is because we will not create new types of Pokemons during the game.

The Pokemon class models Pokemons that are present in the game. Both an ID and an object in PokemonType class are required to instantiate a Pokemon. The ID is required such that any two pokemons owned by the player must be unique. The Pokemon class has fields which covers the basic statistics of Pokemons including HP, CP, attack moves etc. as well as methods such as Hit, Heal, Evolve, PowerUP etc. These methods are called from various functions within the game to update the status of the Pokemon.

The Player class models the player himself. It contains fields such as name, pokeball_count, ownedPokemons, Stardust. This class is useful in storing the status of the Player, as the fields are retrieved by various functions in our game to determine whether the player can perform a particular action, such as evolving a pokemon.

For view-scope classes, we mainly have the CaptureGame class (associated with the Capture view) and the BattleGym class (associated with the Battle view). For the navigation and Manage view, since no complicated logic is needed and most of the logic performed is on updating the view, we did not define additional classes and wrote most of the code in event handlers or timers in the presenter layer.

The CaptureGame Class is associated with implementing the logic used in the mini-game for capturing pokemons. Our game is done by reducing the width of a gray circle and player should press enter key when the circle overlaps with the green circle. We hence have a NextWidth method to return the new width of the gray circle after a fixed time interval. Moreover, we have the checkResult method to return how close the two circle is, the IsSuccessful method to decide whether the Pokemon is successfully captured and the Escaped method to decide whether the pokemon has escaped. The presenter layer will call methods in this class and update the view to reflect the change in status accordingly.

The BattleGym class is associated with implementing the logic in gym battles. This class is fairly simple and can easily model any other turn-based battles (see Class Reuse section). The PlayerMove and OpponentMove methods perform updates on the HP of both parties after making a move, and call the win and lose methods if it detects that a party wins the battle. The win and lose methods have to be passed in as delegates and are not defined in the class, so that it is possible to have different view updates after winning or losing.

## 3. Class Reuse

Our classes feature high reusability in various cases.

The Pokemon class, for example, can be reused in different pokemon games. While other Pokemon-themed games may not feature the same gameplay and require different functions, the Pokemon class is likely to be reusable with minimal changes as they all require similar data (CP,HP, Weight, Height etc.) and similar methods (PowerUP, evolve, Heal etc.). The player class is also highly reusable, as its properties, including pokeball_count, OwnedPokemons, Stardust etc. are also common amongst different Pokemon games.

The BattleGym class we defined might also be useful in any other games featuring turn-based battles. The class models turn-based battles by alternatingly calling PlayerMove and OpponentMove methods, with the win and lose methods not completely defined within the class but requiring users of the class to pass in the win/lose methods as delegates. This allows the class to be reused in a variety of games with only small changes within the class itself.

## 4. Software Patterns used

The Model-View-Presenter model is used when developing the application. It has the advantage of strongly separating the Model and the View, which allows us to develop the two aspects independently.

To apply this pattern rigorously, we avoided starting coding immediately which would likely result in the view and model to be somehow mixed together. Instead, we discussed the plan first and came up with a list of classes to be implemented, each of which performs a specific function or implement a particular game in our application. Then, those classes are implemented first with minimal consideration to the view. This prevents us from unintentionally including code related in updating the view into those classes, which would weaken the separation of model and view.

Moreover, the observer button is used throughout our application. Lots of event handlers are used to trigger the execution of some code when a particular event has happened, for example, clicking some buttons on the interface or typing in some keys. This allows us to easily model roles of each object present in our interface by writing their interaction logic into the event handler.

The strategy pattern is also used in our application. The BattleGym class that we designed makes use of delegates to call the win and lose function when it detects that a Pokemon has run out of HP. The win and lose functions are defined in the presenter module and passed to the BattleGym constructer during instantiation. This allows a higher reusability of the BattleGym class as it can handle different winning or losing logic, e.g. if we want to have a different winning message in different situations, we can simply pass in a different win method when instantiating the BattleGym object.

Finally, we have used the composite pattern several times as it is helps group related objects together and allows us to maintain a good hierarchy of objects. For example, we have used a grid to wrap the message box, which contains a rectangle shape, textblock object and a button together. This makes controlling the message box much more convenient as we just need to set the visibility property of the grid to show or hide the message box, instead of each individual component. It also has the benefit of having clearer logic and cleaner code.

## 5. Challenges Overcome

While most of our development process went smoothly, we did encounter some situations where one of us misunderstood the classes wrote by another person. Sometimes we misused each other's classes and caused the program to not function as expected.

We overcome this problem by using a methodology very similar to pair programming. Although we did not literally sit together and work on the project, we take turns modifying the file and frequently push changes to the GitHub repository. Both of us keep monitoring the changes made and notify each other if we spot some issues in the code. This allows us to be promptly alerted of any errors and avoids having a bug that would possibly take a long time to discover had we waited until each of us finish everything before checking each other's code.

## 6. Work division

| Section | Detail | Wong Keng Lam | Wong Wan Ki |
|---|---|---|---|
| View | Navigation | ✓ | ✓ |
| | Capture | | ✓ |
| | Gym-battle | ✓ | |
| | Manage Pokemon | ✓ | |
| Presenter | Navigation | ✓ | ✓ |
| | Capture | | ✓ |
| | Gym-battle | ✓ | |
| | Manage Pokemon | ✓ | |
| Model | Game Scope classes:<br>Player, Pokemon,<br>PokemonType, AttackMoves | | ✓ |
| | View Scope class:<br>CaptureGame, BattleGym | | ✓ |