IERG3080 Readme

Model, Kovalenko Pavel

1. PokemonModel Namespace.

**PokemonModel** contains the definitions for all the Pokémon, all their information, assigned types, possible attacks and the Pokémon factory. My goal with this class was to automize the Pokémon addition and attack addition, where they can be added infinitely (the attacks need to be scripted).

**PokemonDataLib** – the general data class, short definitions used throughout the model.

**PokemonRecord** – this class is made to describe a blueprint of a Pokémon, which is generated to, for example, show the Pokémon on the map without the need to generate attacks, levels, HP etc. for it. This way I can simply use the builder pattern to make the Pokémon step by step and use the intermediate instances for various things.

**PokemonInstance** – This class is made to describe any Pokémon in the battles. Apart from the general information contained in the PokemonRecord class PokemonInstance Instances also have the level, HP, MaxHP, heavyCharge etc. and the attack delegates. All of these things are used in combat; hence we can describe an enemy Pokémon fully by assigning it with a PokemonInstance.

Each Pokemon instance has two attack delegates, which represent the light attack and heavy attack respectively. Heavy attack deals more damage generally, but only if the heavyChearge is high. heavyCharge is incremented upon every light attack and nullified after a heavy attack.

**PokemonCaptured –** the class, representing the captured Pokémon in the player inventory. Contains the functions, which allow to change the Pokémon, such as GainXP, Rename etc. Every PokemonCapturedClass is assigned with a special number, ID, which is incremented for every Pokémon player catches. This enables many efficient look-up opportunities throughout the code.

Captured Pokemon can also level up in accordance with GainXP and evolve when the level is high enough. These functions are also a part of the PokemonCaptured.

**PokemonDictionary –** this singleton class contains a dictionary for all Pokémon defined in the PokemonInfo.txt file, where the PokemonRecord can be acquired by name and the lists containin all the Pokémon of certain Rarity. Most important is the ability to add Pokémon infinitely to the PokemonInfo.txt and the model automatically accommodating space for them. Additionally, the pokemon evolutions can be defined in the EvolutionInfo.txt can also be manages in much the same way, where it will be handles automatically with a dictionary linking the names.

**AtttackRecord and FullAttack –** these contain information about the attacks, such as names and delegates. AttackRecord is needed for the classes to have all the general information about the attacks, while keeping the delegates private.

**PokemonAttackLib –** this is a singleton class that is storing the attacks together with their corresponding types (Pokémon types and light/heavy), names and base damage. Here, the AttackInfo.txt file is adopted, where attacks can be added infinitely.

**PokemonAttackLog -** this singleton class stores the definitions for all the attack delegates. The flow is as follows. The PokemonAttackLib creates attack entries and looks up the functions contained in the PokemonAttackLog. Then the AttackLib linkes the delegates and the rest of the information creating a look-up interface, which is then used in the pokemon factory to create the PokemonInstances. This is implementing the strategy pattern and passing everything to the factory. So long as the attack is in the AttackInfo.txt and is defined in AttackLog it will be automatically assigned to pokemons with the corresponding types.

**PokemonFactory –** finally, the factory combines all the strategy pattern and singletons into a builder pattern, where the Pokémon are assembles step by step. Given the rarity, the PokemonRecord is generated, based on which the attacks are matched for a Pokémon’s types and then the delegates are passed to the PokemonInstance constructor (which can be used for battles). Finally, when the Pokémon is captured, it is granted with the above-mentioned interface and made into a PokemonCaptured, which can be placed directly in the players inventory.

1. PokemonPresenter Namespace.

PokemonPresenter contains two classes, which are used to exchange the data between the presenter and the model. **PokemonBattle** can be created form a PokemonInstance (for Gym battles) and **PokemonDisplay** is created from PokemonCaptured (for inventory display).

1. PlayerModel Namespace

**Player class** is a traditional singleton, which is used to keep player information, such as the player’s inventory (PokemonCaptured List), link it to the PokemonDisplay list for the presenter and update the presenter on the changes. The Battle-Ready Pokémon list is also managed here, where the list is updated when the player chooses the new Pokémon as battle-ready or the main list is changed. Player class also contains the function OnPokemonChange and OnPokemonAdded, which change the model and report the changes to the presenter.

1. GymModel Namespace.

**GymBattle class** – handles the battle logic and reports the changes to the presenter. Importantly, it only handles the general logic, the rest is handles inside the attack delegates, where the event OnAttackPerformed reports the data to the presenter as PokemonBattle instances and HP of the attacked Pokémon. The class itself is a singleton with the private instance access, which is created on OnBattleStart event and deleted when the battle is over. This ensures that there is no possible misuse of the class outside the battle, as the instance cannot possibly be accessed. This class leverages the delegate nature of the attacks by calling the directly on PokemonInstance instances inside.

1. MapModel NameSpace.

**Map –** a traditional singleton class, which contains navigation data. Global Map is the map of maps, which contains a list of map Ids for every location. GlobalMap list is expanded with random (unique to its neighbors) map IDs, hence making sure that the map generation is random, but the map is stored in memory. Expansion is dynamic, so the Global map only keeps the IDs depending on how far player went from the start. Pokémon spawns are defined in a dictionary, which links location tuples to Pokémon names, therefore, if the new location is in the dictionary the Pokémon is stepped on. There are timers between Pokémon spawns to prevent over spawning.

1. CaveModel and MazeGen Namespaces.

**Cave –** Is very similar in structure to the GymModel and is also a private singleton. The mechanics are similar to the MapModel, only much simpler and without the need to change maps. Player is rewarded a Pokémon with bigger rare chances upon winning.

**Path class** – contains methods, which allow to recursively verify is there are possible paths on a given labyrinth between the two given points.

**Generator Class** – contains methods to take the given (largely unfinished) maze preset and with the use of Path class, by choosing points on the map which have to have to be accessible from start, randomly fill the labyrinth with walls. This method of generation makes mazes special every time, while keeping things simplistic and not requiring too many assets.

End Results: Model designer can simply fill the .txt files with the necessary information and define it on their side. In the end of the work, there is no need for the presenter to know anything about the model code, as everything can be manipulated using the event data. Communications are done in full through events between the model and the presenter. To accomplish this I have used Strategy pattern for convenient attack management and handling, Builder pattern for simple and manageable Pokémon construction and Singletons for maintaining independence of the model classes.

View, Zheng Weijia

1. MainWindow

**MainWindow** is the title screen of the whole game, while its implementation is relatively simple. And it does not use any model mentioned above.



2. NaviMap

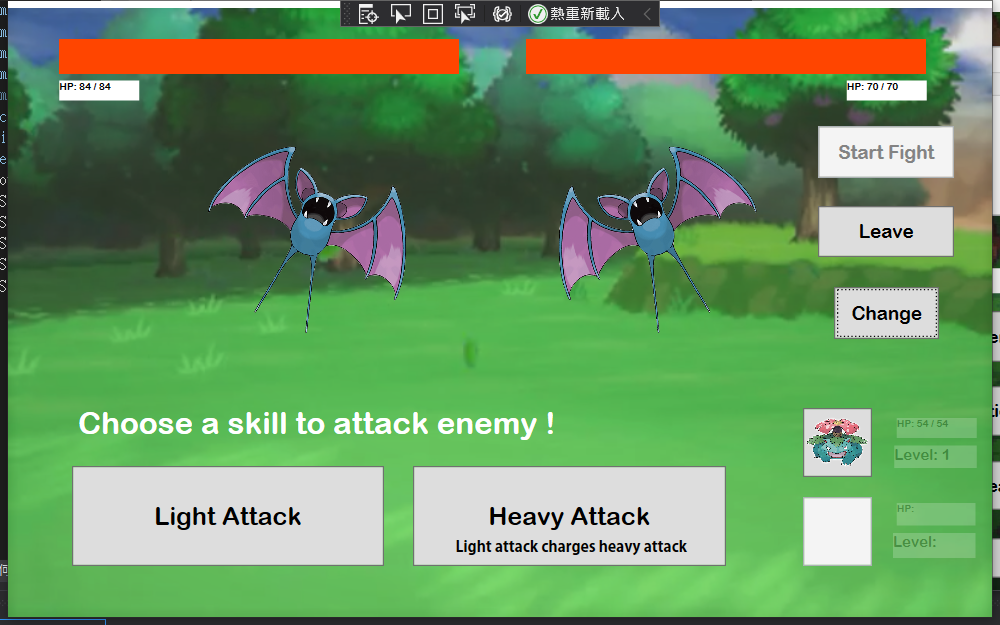
**NaviMap** is the main part and the most important part in the View. Players can Manage his Pokémon’s here, e.g., feeding Pokémon’s with experience potions, **Rename** their nicknames, view their conditions, choosing some Pokémon’s to the battle team and even selling them for more experience potions. This part quotes most previously mentioned namespaces and need to update the views when player interact with his Pokémon’s. (Although well done the Model side by Pavel, I do not have enough time to replace some of my early-stage codes, which are not cooperating with the Model very well, so there may exist some minor issues.)



Besides, here one can also control the player, by pressing the W, A, S, D on keyboard or on the screen, to let the main character move in four directions. This part obviously uses the **PlayerModel** and the **MapModel**. To achieve movements, at the view side it is relatively easy because only need to call the functions in Model and use some events. To draw the map, which includes three layers of thing (main character, Pokémon’s and environment things) I used three layers of wrap panels and use squares to fit in them. There are both events when stepping on a gym, a cave, or a Pokémon. We’ll overview them one by one.

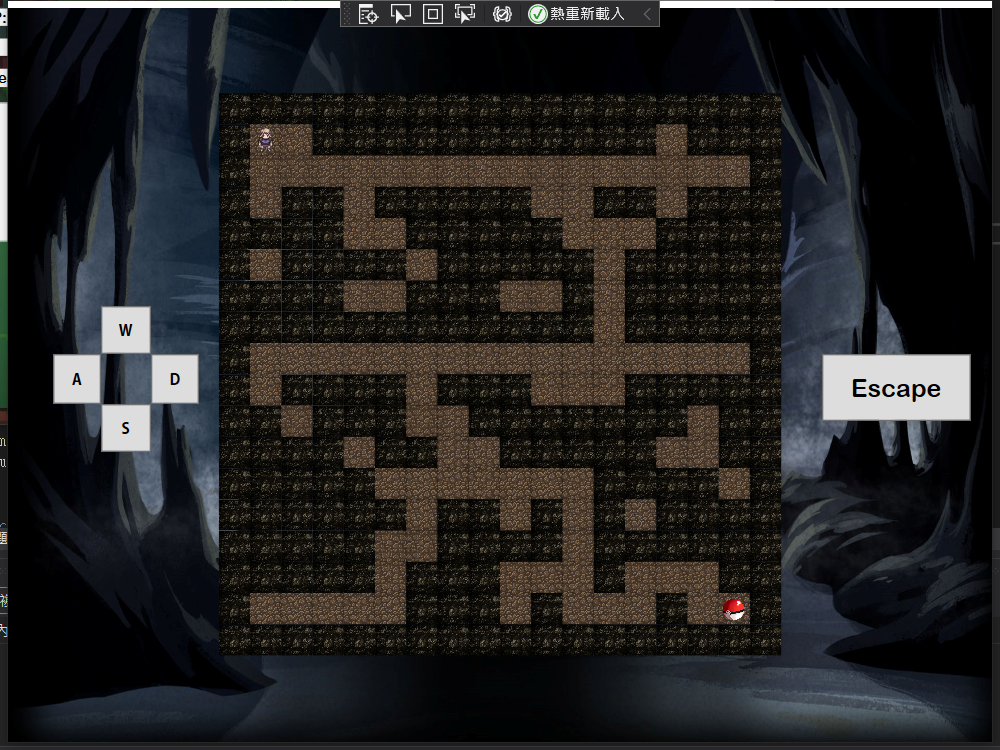
3. GymBattle

**GymBattle** is the view for a battle in a gym. Considering Models. The implementation of this part should be easy. One can use at most three of his Pokémon’s to challenge a gym (which contains 2 enemy Pokémon’s) to gain experience value. Not only can one use light/heavy attacks to hit enemy, but can he change another Pokémon in his battle-ready-list in the mid-way.



4. MazeCave

**MazeCave** is our group’s extra feature, which is purposed to the Piazza at an early stage. Once can go through the maze to get reward, the hard part of this part is the generation of maze, which is well handled by the Model part in **CaveModel**.



5. Capture

**Capture.xmal** will be shown once you stepped on a Pokémon. And a typing game is set for the player to complete if he wants to capture the Pokémon. One needs to correctly type the name of the Pokemon shown on the map (also now shown on the Capture window) to capture it, otherwise it will run away.

