1 Overall Design

The Model-View-ViewModel (MVVM) is adopted in the project as our design pattern. That means each View is bound with a ViewModel, and there is a Model to store business logic and data. In WPF, MVVM design pattern is achieved using data binding, so that each View retrieves binding data from its ViewModel and updates itself without knowing and performing any logic. Therefore, presentation logics and data are stored inside ViewModels which are responsible for signalling Views in case of value change of data. Since "Binding" objects are implemented inside Views, ViewModels must also implement their "INotifyPropertyChanged" interface, in order to make use of the event handler and fires the "PropertyChanged" event to inform Views specific property change. Due to this special relationship between Binding class inside View and fields inside ViewModel, it is convenient to define the property for each field and inject the "OnPropertyChanged" method to the setter of property, such that if there is any field assignment, a "PropertyChanged" event is automatically fired and "Binding" objects inside View know they should update their value by looking at corresponding properties. The following figure shows the inheritance tree for all classes defined in the program. One may find that all Models and ViewModels are able to use "OnPropertyChanged" method to fire the event. On the other side, methods inside ViewModels are also bound with Views, so that users are able to invoke them from Views and let ViewModels perform all presentation logics and update properties.

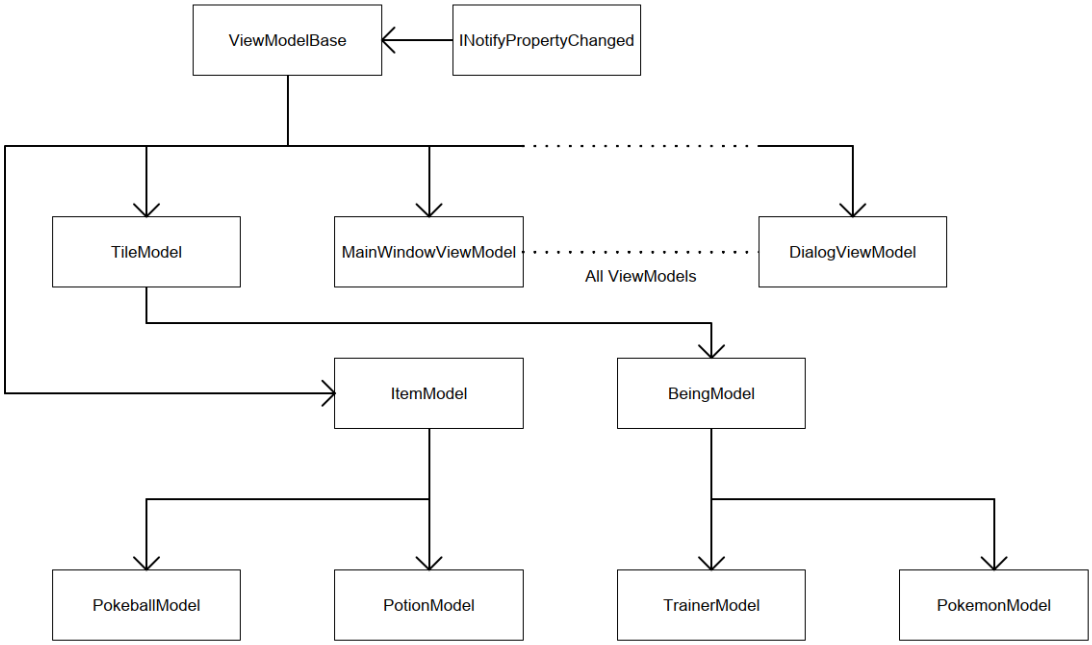

Figure 1: Inheritance Tree

For example, inside each ViewModel, all the fields are private and designed to be accessed using a public property so that the "OnPropertyChanged" method is included inside the setter and it is always called if there is assignment. Moreover, Models also inherit ViewModelBase in order to provides easy controls.

2 Design Pattern Disadvantages

Though MVVM design pattern provides various advantages such as separation of user interface and application logic, such that application is easier to be maintained and evolved, these may turn out to be its disadvantages. As shown in the following figure 2, Views of application are separated, and they do not communicate with others. Unlike MVP or MVC design patterns which make use of code-behind to control window popups and views navigation, Views of MVVM provide limited code-behind methods which controls additional windows popup and change of views, in order to achieve the goal of separation.

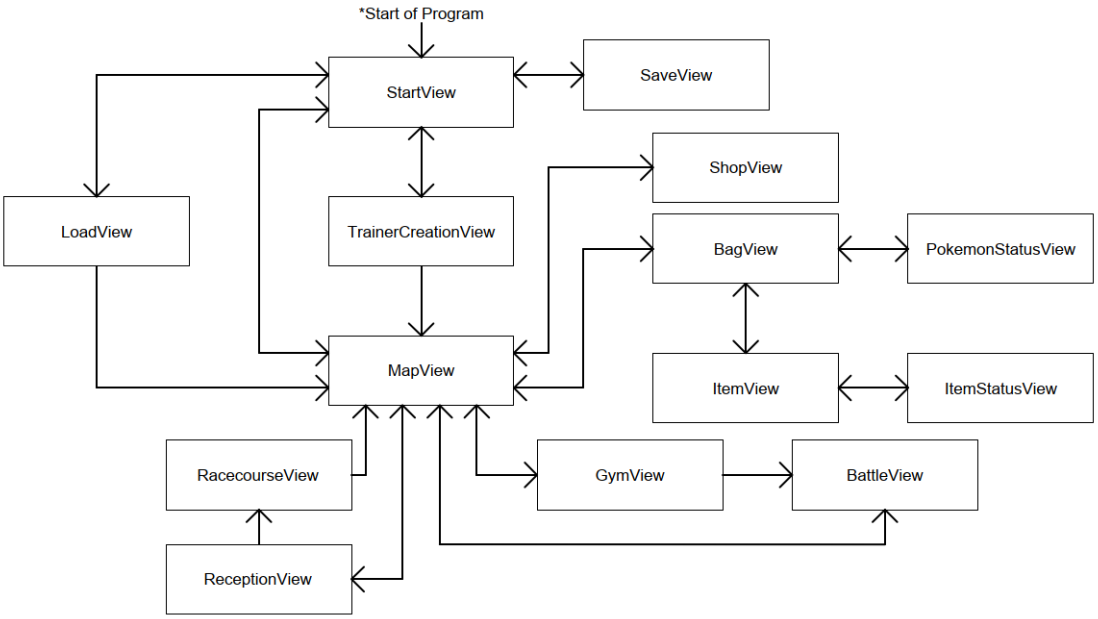

Figure 2: Flow of Program

In order to resolve the problem, a "MainWindowViewModel" class is designed to be the primary ViewModel which is responsible for views navigation. All other ViewModels are declared and initialized inside "MainWindowViewModel" which is able to "pick" one of these to be the "CurrentView" and "CurrentViewModel". Therefore, content displayed by the application window is always bound with "CurrentView" and "CurrentViewModel". All other ViewModels are eligible to call class methods (which is illustrated in the later part of this report) to tell "MainWindowViewModel" to change the view. The following figure 3 shows that sliding window analogy of this design. The main window of application is always showing the "CurrentView" which is bound with "CurrentViewModel", and "MainWindowViewModel" responses to any method call of changing "CurrentView" and "CurrentViewModel", by sliding on the ViewModel series. This feature is implemented using backlink such that "MainWindowViewModel" is parent of all other ViewModels and all other ViewModels are able to access his parent's fields and methods. Therefore instead of passing data between ViewModels, data are linked and reference using backlinks so that any field of any model is accessible by any model.

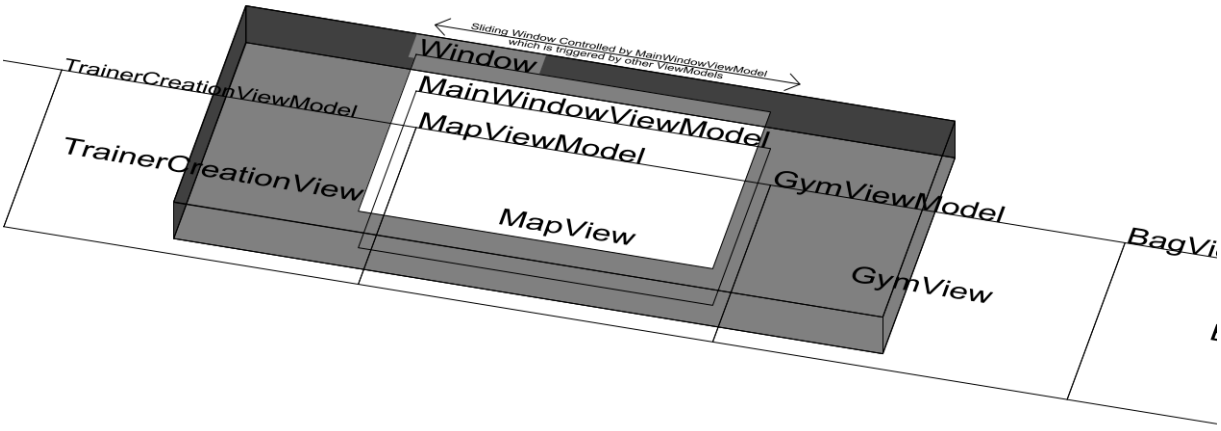

Figure 3: Analogy of Sliding Window Implemented Using MainWindowViewModel

Another challenge of adopting the MVVM design pattern is to simulate the behaviour of "System.Windows. MessageBox" class which is often used to display pop-up windows for MVC or MVP design pattern. Since ViewModels are expected to know nothing on how to display and format the presentation style so that it is capable of design change and evolution (such as the cancel additional requirement), the implementation of dependent display logic such as "MessageBox" class is

considered as a bad practice. Therefore, a “DialogViewModel” is designed to resolve the problem. Pop-up messages of string are stored inside, and Views are able to bind properties of model to show overlaying dialog box if it is needed.

3 Key Features Highlight

3.1 Battle

The battle of the game inherits from the original Pokemon Black and White. Player is able to fight other Pokemons with his Pokemons which have unique set of abilities. There are 3 types of battle namely NPC battle, gym battle and wild Pokemon battle in this game. Although use of items is allowed during battle, item “Pokeball” is only allowed for wild Pokemon battle.

3.2 Capture Wild Pokemon

Pokemons are randomly spawned in the map, and player is able to capture with 3 types of ball namely normal, great and master ball. They increase the chance of successful catch by 20%, 50% and 100% respectively.

3.3 Gym

If the gym is occupied by somebody, player can challenge it at any time. If the gym is occupied by the player, someone will randomly challenge the player who needs to defend by accepting the battle. Otherwise, the gym may probably drop.

3.4 Extra Features

3.4.1 Racecourse and Reception

Reception provides refreshment to restore all player’s Pokemons regularly. In addition, player is able to enjoy the Pokmeon racing after placing his bets.

3.4.2 Shop

A shop allows player to buy Pokeballs and potions. There is also a special corner for player to randomly draw a Pokemon.

3.4.3 Save and Load

This is initially implemented for debugging. Since data are stored into a single list and it is easy to be serialized and desterilized. Player can save the game to and load the game from the binary file.

4 Classes Highlight

4.1 ViewModelBase

As illustrated before, this is the base class of almost all classes. A “OnPropertyChanged” method is implemented so that any subclass is able to fire “PropertyChanged” event.

4.2 Models

4.2.1 TileModel

TileModel is inherited by all objects which are related to MapView since the map is constructed with tiles. Field of 2D coordinates are defined such that each object knows its position on the map.

4.2.2 BeingModel

BeingModel inherited from TileModel which is an abstract class for all beings in the game. Being properties such as name and health are defined.

4.2.3 PokemonModel

PokemonModel inherited from BeingModel. Unique fields such as “Abilities” are defined.

4.2.4 TrainerModel

TrainerModel inherited from BeingModel. Unique fields such as “Pokemons” and “Items” are defined.

4.2.5 AbilityModel

Basic properties such as damage and heal power of ability are defined. A game data list of “AbilityModel” is initialized at the start of application by looking at the predefined json file. Special effects are also implemented.

4.2.6 ItemModel

It is an abstract class storing the field of different items with abstract method Use which actually is a Template Method. So that when we add a new kind of item that needs to be added, we can just inherit from this class.

4.2.7 PokeballModel

It is a class inherited from ItemModel to store the Pokeball, in other words, Pokeball is an Item. Therefore, it contains an override method “Use”.

4.2.8 PotionModel

It is a class inherited from ItemModel to store the Potion, in other words, Potion is an Item Therefore, it contains an override method “Use”.

4.2.9 LogModel

This Model is used to wrap log message and log Id as a whole.

4.3 View

The implement of view remains sample under MVVM design pattern. Code-behind contains no presentation and navigation logic, and controls inside view are responsible for invoking “Binding” methods. Attribute values of controls are “Binding” objects which listen to “PropertyChanged” event to retrieve data and update.

4.3.1 MainWindowViewModel

As illustrated before, “MainWindowViewModel” which is the “most important” ViewModel, mainly controls which View is displayed and which ViewModel is used. Methods of GoToXXViewModel are invoked and control what is the CurrentViewModel and CurrentView. Since it is the parent all other ViewModels, data linkages and references are constructed via MainWindowViewModel.

For example, one can invoke UpdatePlayer method of GymViewModel even if he is outside GymViewModel by calling (GymViewModel)MainWindowViewModel.GymViewMolde.UpdatePlayer(Player);. one shall call to update the Player inside the GymViewModel, in order to create the reference and backlink.

4.3.2 BagViewModel

This ViewModel contain UpdatePlayer Method to let other ViewModel to update the field inside the ViewModel.

4.3.3 PokemonStatusViewModel

There is an UpdateView method to update the ViewModel. When the player wants to sell a pokemon, it will call SelPokemon. It will call the drop pokemon method in Trainer Model. If a player wants to change name, it will be checked by the private method “ChangeName”. If it is empty, it will be replaced by the original name. For each action made in the PokemonStatusViewModel, it will call DialogViewModel, which controls the pop out in run time.

4.3.4 ItemViewModel

This is a similar ViewModel as BagViewModel only the item is different.

4.3.5 ItemStatusViewModel

There is an UpdateView method to update the ViewModel. When the player wants to drop an item, it will call DropItem. The drop item method is to call the TrainerModel.

4.3.6 BattleViewModel

There is AI implemented and it simulates a normal player behaviour. For example AI always tries to heal his Pokemon if it is in danger.

4.3.7 DialogViewModel

In this ViewModel, it contains a delegate to control which method will be used when we press on the “ESC” or “OK” button. It is basically using Strategy.

4.3.8 MapViewModel

There is a timer to Control when the player occupies the gym. MapViewModel also mainly control the game, therefore, at most of the time, when we need to changeView, we will update the field by calling the update___ Method inside each of the ModelView.

4.3.9 ShopViewModel

In “ShopViewModel.cs”, inside the method “RandomPokemon()”, there is a line “PokemonModel pokemon = RandomPokemonMethod();”. The RandomPokemonMethod() is a factory method to randomly generate and initialize the Pokemon to the field “pokemon” without having logic inside the method “RandomPokemon()”.

4.3.10 ReceptionViewModel

Player is able to have free refreshment regularly.

4.3.11 RacecourseViewModel

By simulating the racing using Pokemons health, Pokemons are racing to gain a target as soon as possible.

5 Division of Work

Type of Work	*Contribution*	
	Lam Ka Shing	***Kwok Ka Chun***
1. Model		
1.1. AbilityModel	✓	
1.2. BeingModel	✓	
1.3. ItemModel		✓
1.4. LogModel	✓	
1.5. PokeballModel		✓
1.6. PokemonModel	✓	
1.7. PotionModel		✓
1.8. TileModel	✓	
1.9. TrainerModel	✓	
2. ViewModel		
2.1. BagViewModel		✓
2.2. BattleViewModel	✓	
2.3. DialogViewModel	✓	
2.4. GymViewModel		✓
2.5. ItemStatusViewModel		✓
2.6. ItemViewModel		✓
2.7. LoadViewModel	✓	
2.8. MainWindowViewModel	✓	
2.9. MapViewModel	✓	✓
2.10. PokemonStatusViewModel		✓
2.11. RacecourseViewModel	✓	
2.12. ReceptionViewModel	✓	
2.13. SaveViewModel	✓	
2.14. ShopViewModel		✓
2.15. StartViewModel		✓
2.16. TemplateViewModel	✓	
2.17. TrainerCreationViewModel	✓	
2.18. ViewModelBase		✓
3. View		
3.1. BagView		✓
3.2. BattleView	✓	
3.3. GymView		✓
3.4. ItemStatusView		✓
3.5. ItemView		✓
3.6. LoadView	✓	
3.7. MainWindow	✓	
3.8. MapView	✓	
3.9. PokemonStatusView		✓
3.10. RacecourseView	✓	
3.11. ReceptionView	✓	
3.12. SaveView	✓	
3.13. ShopView		✓
3.14. StartView		✓
3.15. TrainerCreationView	✓	
4. Game Data Design (.json)		
4.1. Abilities		✓
4.2. Items		✓
4.3. Pokemons		✓

4.4. Trainers	✓	
5. Debugging and Testing		
5.1. Battle		✓
5.2. Capture Pokemon		✓
5.3. Shop		✓
5.4. Racecourse		✓
5.5. Load and Save	✓	
5.6. Branch pre-released-1.0	✓	
5.7. Branch pre-released-1.1		✓
5.8. Branch released-v1.0		✓
5.9. Branch released-v1.1	✓	✓
5.10. Branch released-v1.2	✓	✓
5.11. Branch released-v1.3	✓	✓