IERG3080 Project Part 2 Report

1155116155

HEGAZY Mahmoud Ahmed Mahmoud Mohamed M.

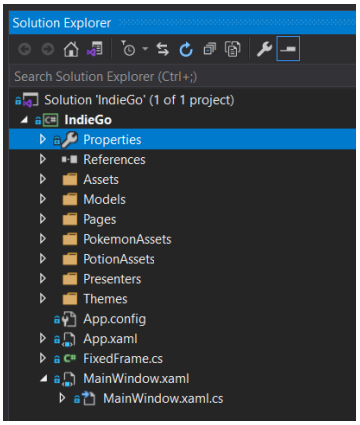
Introduction:

This is the project report for project part 2 IERG3080. It illustrates the design choices and patterns implemented in the project and how they constitute good software engineering practices. Throughout this document, the focus is to explain how different design patterns and architectures were implemented and why they were chosen, and their relative advantages to alternatives.

The first section describes how the code repository is organized and structured. This is to allow the reader to have an easier time reading the code and understanding its different components' functionalities. The second section aims to illustrate the architectural pattern used in this project, which may be closer to MVP-MVVM hybrid pattern. Subsequently, the third section will go into more detailed creation and implementation patterns and how they were used to design different classes. This section will also demonstrate how the current architecture can adapt to changes in requirements.

Section I: Code Organisation

From a bird's eye view, the repository comprises the following seven folders (Assets, Models, Pages, PokemonAssets, PotionAssets, Presenters, Themes). The folders Assets, PokemonAssets, and PotionAssets are responsible for storing different images responsible for the GUI components and do not contain any code. It may be helpful to note that potions were implemented as an extra feature. During the gym battle, they can improve pokemon attack or defense or/and the money gained upon winning.



The folder Themes contains a single file, "Generic.XAML". This file is a default WPF file used to control the style and implementation of custom controls. The game only utilizes one custom control, "FixedFrame," which is implemented in "FixedFrame.cs". As a result, Themes and "Generic.XAML" are minimal components, and not reading them would not impact the understanding of the remaining codebase.

The folder Models contains the core logic of the game and its most important classes. The main classes in this folder are Player, Pokemon, and PotionBase. Pokemon and PotionBase are the base classes that act as templates for pokemon and potions. Player is a relatively simple class that contains the player information, which consists of 2 ObservableCollections for potions and pokemon, stardust balance, and pokedollar balance. Some of these classes implement the INotifyPropretyChange interface to allow for dynamic data binding. Furthermore, Models contains the implementation of different types of pokemon and potions inheriting the template classes. Finally, the Models folder contains some other classes to create a random potion or a pokemon.

The Presenter folder contains the three presenters responsible for three primary states of the game: GymModel, InventoryModel, and MiniGame. InventoryModel is accountable for the Inventory "Pokémon Lab". GymModel and MiniGame are responsible for the gym battles and catching Pokémon games, respectively.

Finally, the Pages folder is where the actual UI of the game exists. Four pages represent the game start screen, main map, gym battles, inventory/Pokémon lab. The "code behind" of these XAML files is relatively small as much of the game logic is dedicated to the .Presenters and .Player namespaces. However, some minimal logic related to the gamified nature of the presentation is there.

Section II: Architectural Pattern

The main architectural pattern used in this game is a hybrid MVP-MVVM. This pattern was mainly chosen as it provides a suitable trade-off between decoupling of entities, code overhead, and

complexity. To motivate and explain the architecture used, I will briefly go over MVC, MVP, and MVVM main advantages and disadvantages. MVC does not guarantee the decoupling between the model and the view. This would have produced less readable code that is also less flexible to change as editing view requires editing the model. MVP goes a step further and guarantees a decouple between the model and the view, which is an exceedingly essential feature. However, MVP's main problem is the interaction between the presenter and each of the two other entities. This might become problematic as it requires changes in the model to be implicitly propagated back to the view through the presenter. This restricts the flexibility in changing the model.

On the other hand, MVVM goes a further step by guaranteeing complete decoupling of Model, ModelView, and View. It is arguably the most natural architecture to implement in WPF, which was initially designed to work with it. WPF provided many tools to implement MVVM, such as ICommand interface, data binding, and INotifyPropertyChanged interface. The first provides a mechanism to allow View elements to trigger custom events inside custom classes. The second makes UI elements display property of objects, while the last allows dynamic data binding, i.e. UI elements change as properties are updated. While this is ideal for separation, it requires coding overhead and increased complexity.

The architecture implemented in the project can be thought of as MVP with data binding. More precisely, the presenter contains multiple Models objects and provides some operations involving them. These models' instances are represented as properties of the presenter with data binding to the view. As a result, changes to the model propagate back through data binding leveraging the INotifyPropertyChanged interface. This provides complete separation between the model and view and further decreases the dependence between the view and presenter. However, the overhead of implementing the ICommand interface is avoided. This is the most significant overhead of MVVM in game-like applications

It must be noted that the definitions of MVC, MVP, and MVVM may slightly differ across resources. Moreover, the lines between different patterns may be further blurred in practice. For example, I described the architecture as a hybrid MVP-MVVM because it implements MVP with data binding influence from MVVM. However, such characterization may not be correct under some very specific definitions of MVP and MVVM.

Section III: Design Pattern

Template Pattern

The template pattern is one of the most useful and widely used patterns. The classes Pokémon and PotionBase provide the templates for all types of Pokémon and potions. This provided multiple benefits, including code reusability and extensibility. For example, the Pokemon class applies the INotifyPropertyChanged to allow dynamic data binding as the instance is evolved or powered up. This eliminates the need to implement the interface in every pokemon consistently. In the implementation of this pattern, the SOLID principles were minded. Specifically, the template classes apply the open-close and dependency inversion principle. For instance, the pokemon class provides methods such as AbsorbArrack, GetAttack, and Heel, which allow the class user to use the pokemon in battle without knowledge of the class's internal mechanisms.

Singletons Creation Pattern

The singleton pattern was used numerous times in the project, such as in the Player, InventoryModel, PokemonGenerator, and PotionGenerator classes. The latter two were implemented as static classes, but they may be considered singletons from an abstract perspective. This creation pattern was extremely beneficial in guaranteeing only one Player instance and allowing access to it from different places. The InventoryModel was chosen to be created as a singleton because the logic does not change through execution. Thus, it was a safer and probably more optimized design decision to allow only one instance.

Adapt to Change:

As main design choices are now illustrated, it is possible to imagine how this design can adapt to future changes. Possible scenarios are mentioned below.

1. Add new Pokemon species

Due to the leverage of template patter adding new pokemon is very easy and does only require few lines of code. To add a pokemon species: 1) Inherit the template class and implement Evolve similar to other build classes. 2)Add the assets to the folder Pokemon Assets, 3) Edit PokemonGenerator.Get() to add the constructor of your new pokemon

2. Change of the Main Map/ Inventory Layout

The main map does not hold any game logic in it. Its only function is to trigger going to the gym, inventory, or catch game. As a result, the main map can be changed easily to other formats as long as they possess a way of triggering page transfer. This is due to the design pattern allowance for modularity.

The Inventory is mostly implemented using data binding. Hence, the display can be changed into any shape without affecting the presenter. Besides, InventoryModel automatically adapts to changes in the types of pokemon and their characteristics.

3. Change of the format of Gym Battle

This is probably the most demanding of the changes to implement as a Presenter logic needs to change. However, the code that needs to change remains relatively small as only two files will need editing, with the remaining of the game being untouched. This demonstrates the independence between different parts of the game, allowing each state to behave like Lego that can be added to removed.