2110521 Software Architecture

# **Assignment 3:** MVC, MVP and MVVM Architectural Pattern

## Team member information:

|  |
| --- |
| **Team Name:** four-guys-one-cup  **Member 1:** Nuttanai Kijviwattanakarn (6030200821)  **Member 2:** Time Yongyai (6030285121)  **Member 3:** Wattanai Tipsathaporn (6030515521) |

## Objective:

1. To understand the concept of patterns for achieving the separation of concerns in software design
2. To understand the concept of Model-View-Controller pattern
3. To understand the concept of Model-View-Presenter pattern
4. To understand the concept of Model-View-ViewModel pattern

## Requirement:

1. Python 3.7 or greater
2. wxPython for UI development (<https://www.wxpython.org/>)
3. RxPY for reactive programming (<https://rxpy.readthedocs.io/en/latest/index.html>)

P.S. The program in this assignment is designed to run on Windows, macOS and Linux.

## How to submit:

1. Create your new group repository in the class organization with all of your source code
2. Answer each question in this document
3. Submit the document with your answers and your repository link in myCourseVille

## Before we start:

When developing software, usually, the presentation layers (GUI/CLI/etc.) and business logic layers are included. There are many ways to communicate between these layers. The easiest way for a presentation layer is to access business logic directly. Alternatively, you can introduce another layer between these layers which may be better for separation of concerns design principle.

Q1: What is separation of concerns?

|  |
| --- |
| Seperation of concern is a design principle to separate a program into distinct sections that each section addresses a separate concern so that manipulating one section won’t affect other sections. |

Q2: Do you think that we should access the business logic layers directly from presentation layers? Why?

|  |
| --- |
| No, because if there are any change in presentation layers or new view implementation, you need to implement the business logic code to support them though you may have some code logic that can work with them but that code is bound only to work with other view and that results in code duplication and also unnecessary complex code. |

Now, we will setup the development environment for this assignment

1. Install wxPython

|  |
| --- |
| # if you are using Windows of macOS  $ pip install -U wxPython  # if you are using Linux  # Method 1: build from source  $ pip install -U wxPython  # Method 2: Find binary suited for your distro  # For example with Ubuntu 16.04  $ pip install -U -f <https://extras.wxpython.org/wxPython4/extras/linux/gtk3/ubuntu-16.04> wxPython  # Method 3: Find the package in your distro repository |

1. Install RxPY

|  |
| --- |
| $ pip install rx |

P.S. In some OS, “pip” command is pointed to the pip for Python 2. Anyway, we used Python 3 here so please make sure to use the correct “pip” command since it might be “pip3” in some environment such as macOS.

Next, clone the provided git repository

|  |
| --- |
| # Change directory to your desired directory  [your directory] $: git clone https://github.com/2110521-2563-1-Software-Architecture/Assignment-3-MVC-MVP-MVVM.git |

## MVC: Model-View-Controller

Firstly, we start with MVC pattern. In this pattern, three components are presented including Model, View and Controller. The model includes all of your business logic, the view includes all of your program presentations and the controllers are places where your views interact with the models. Note that the implementation of MVC has many variants in addition to the way used in this assignment.

View

Controller

Model

Figure: The MVC pattern and the interaction between each layer

We will create the simple note taking application as shown below

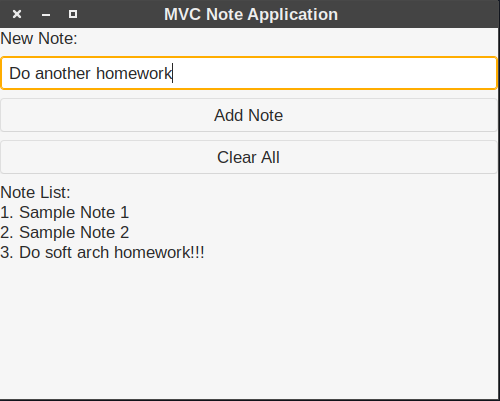


Figure: The simple MVC Note Application which we are going to create

Change directory to the “mvc” folder

|  |
| --- |
| # From assignment root  $ cd mvc |

You will see these structure

mvc

├── main.py

└── mvc

├── controllers

│ ├── \_\_init\_\_.py

│ └── main\_controller.py

├── \_\_init\_\_.py

├── models

│ ├── entities

│ │ ├── \_\_init.py

│ │ └── note.py

│ ├── \_\_init\_\_.py

│ └── repositories

│ ├── \_\_init\_\_.py

│ └── note\_repository.py

└── views

├── base\_view.py

├── \_\_init\_\_.py

└── main\_view.py

What we already implemented for you is all the presentation parts and business logic parts, the objective task is to implement the controller and connect the views and models through the controller.

To run the application

|  |
| --- |
| $ python main.py |

Also make sure that the “python” command pointed to Python 3. If it pointed to Python 2, use the command “python3” instead.

Open main\_controller.py and implement all missing methods.

**Hint:** Use an object of type NoteRepository to interact with the business logic.

Q3: How did you make the controller work?

|  |
| --- |
| We implement the code in way that when we instantiate a controller, it will instantiate a NoteRepository object as its property. After that, we implement methods of controller using a note\_repository property’s method like get\_all\_notes() which call NoteRepository.get\_all\_notes(). |

Next, we will connect our views to the controller. Open main\_view.py and implement all missing methods.

**Hint:** Use an object of type MainController which you implemented in the previous step.

Q4: How did you make the view work?

|  |
| --- |
| We implement the code in way that when user interact with the UI, UI will call View’s method. Inside that method, it will call controller’s method to do the business logic follow received action from user and after that the View will update itself with the return manipulated value from controller. |

Q5: What is the role of the controller here? Explain it breifly.

|  |
| --- |
| Controller acts as a brain of the entire MVC system to connect view and model and manage data flow between them. Controller will receive an action from view, then compute and manipulate models follow the action. |

Q6: What are the advantages of MVC pattern?

|  |
| --- |
| MVC is easy to implement and support fast and parallel development because the modification and implementation on one module won’t affect other modules. And also, In MVC, you can create multiple views for a model, while code duplication is very limited because it separates data and business logic from the display. |

Q7: Put the screenshot of the MVC Note Application displaying your members’ name in each note.

|  |
| --- |
|  |

## MVP: Model-View-Presenter

Now, we will look into another pattern called “MVP” which is considered as a variant of MVC pattern. In the MVC pattern, the view needs to update itself when the data changes which may not be convenient in a complex application. Instead, we will replace the controller with the presenter and change the way they communicate to each other. In MVP pattern, the presenter will be the object which updates the view instead of the view itself.

View

Presenter

Model

Contract (interface)

Figure: The MVP pattern and the interaction between each layer

According to the figure, notice that the view and presenter don’t directly communicate to each other but through an interface (We use simple class here for this assignment since Python doesn’t have the interface).

Q8: In your opinion, why does an interface need to be introduced between the view and the presenter?

|  |
| --- |
| To loose coupling between presenter and view |

Change directory to the “mvp” folder

|  |
| --- |
| # From assignment root  $ cd mvp |

You will see these structure

mvp

├── main.py

└── mvp

├── contracts

│ ├── \_\_init\_\_.py

│ └── main\_contract.py

├── \_\_init\_\_.py

├── models

│ ├── entities

│ │ ├── \_\_init.py

│ │ └── note.py

│ ├── \_\_init\_\_.py

│ └── repositories

│ ├── \_\_init\_\_.py

│ └── note\_repository.py

├── presenters

│ ├── base\_presenter.py

│ ├── \_\_init\_\_.py

│ └── main\_presenter.py

└── views

├── base\_view.py

├── \_\_init\_\_.py

└── main\_view.py

We will start by implementing the contract. Open the file main\_contract.py

Our view needs to be updated by the presenter, to achieve this, the view needs an update method exposed through its interface.

Add these methods to the MainContract.View class

|  |
| --- |
| def update\_view(self, items: List[Note]):  pass |

In the same way, the presenter is also accessed by the view therefore we also need to provide required methods in the MainContract.Presenter class

Add these methods to the MainContract.Presenter class

|  |
| --- |
| def add\_note(self, note: str):  pass  def get\_all\_notes(self):  pass  def clear\_all(self):  pass |

Notice that both MainContract.View and MainContract.Presenter were extended from their corresponding base class.

Next, we will move to the presenter. Open main\_presenter.py, you will see the MainPresenter which is the implementation of MainContract.Presenter.

The MainPresenter should implement all of the required methods stated in its interface. You will need to write these methods to interact with the business logic.

**Hint:** The view also passed in the constructor with type MainContract.View. That’s the way the presenter updates the view. Also, don’t forget to update the view when the data changed.

Q9: What is the role of the presenter?

|  |
| --- |
| Listening events from Model and Views. Then, interact actions between them. |

Q10: What is the main difference between the method in the MainController of the previous section and the method which you just implemented in the MainPresenter?

|  |
| --- |
| In MainPresenter, we implement presenter to call view to update. However, in MainController, we didn’t implement that, we let view update itself. |

The next part is to implement the view so that it can interact with the presenter. If you look at the base\_view.py, you will notice the set\_presenter method which is called by the BasePresenter constructor. This way both view and presenter are now seeing each other.

Open the file main\_view.py and implement all missing method implementation.

**Hint:** Your code only needs to interact with the presenter. Also don’t update the view inside the view and let the presenter do that.

Q11: How did you interact with the presenter? Do you think it makes the implementation of view harder or easier? Why?

|  |
| --- |
| just call function of MainPresenter.  easier because view doesn’t have update itself when data changes. |

Q12: Put the screenshot of the MVP Note Application displaying your members’ name in each note.

|  |
| --- |
|  |

Q13: What are pros and cons of MVP pattern compared to MVC pattern?

|  |
| --- |
| Pros: View is more loosely coupled to the model. Easier to unit test because of interface used.  Cons: It is more complex in terms of coding. Furthermore, presenter might become too big and complex. |

Q14: With MVP pattern, do you think that your application is more testable? Why?

|  |
| --- |
| Yes, it does.  Because, To interact with view layer, it has to pass through the interface. |

## MVVM: Model-View-ViewModel

Next, we will look into the MVVM pattern. In this pattern, we incorporate the reactive programming paradigm in which we make the view update itself automatically when the data change. This can be achieved by letting the views to act as observers while the view model serves the observable stream needed for the UI.

In this assignment we use RxPY, the ReactiveX library for Python, to do reactive programming. You can investigate the RxPY documentation at <https://rxpy.readthedocs.io/en/latest/> and ReactiveX at <http://reactivex.io/>.

Q15: What is reactive programming?

|  |
| --- |
| Reactive programming is the programming concerned itself with asynchronous stream of data and its changes by using observable sequences instead of order of the code. |

Q16: What is the observer pattern?

|  |
| --- |
| It is one of software design pattern in which the one called subject, maintains list of observers, and automatically notify them of any state changes. |

In this pattern, the controller is replaced by the view model. The view model serves the observable stream. The view owns the view model as a field and then subscribes to it so that the view knows when to update itself.

One advantage is that the view model is totally decoupled from the view which also means that you can use it with other views without changing the content in the view model.

View

ViewModel

Model

Figure: The MVVM pattern and the interaction between each layer

According to the figure, notice that the arrow from ViewModel to View is a dashed line. This is because the view doesn't interact with the view but the view just observes the change in view model.

Q17: Do you think that the view model should know which view object is owning it? Why?

|  |
| --- |
| No, because the View Model does not actually interact with the object owning it. The object owning the View Model merely observes the change in the View Model. |

Change directory to the “mvvm” folder

|  |
| --- |
| # From assignment root  $ cd mvvm |

You will see these structure

mvvm

├── main.py

└── mvvm

├── \_\_init\_\_.py

├── models

│ ├── entities

│ │ ├── \_\_init.py

│ │ └── note.py

│ ├── \_\_init\_\_.py

│ └── repositories

│ ├── \_\_init\_\_.py

│ └── note\_repository.py

├── view\_models

│ ├── \_\_init\_\_.py

│ └── main\_view\_model.py

└── views

├── base\_view.py

├── \_\_init\_\_.py

└── main\_view.py

Now, open main\_view\_model.py and implement all the missing things.

**Hint:** Look at <https://rxpy.readthedocs.io/en/latest/reference_subject.html>.

Q18: How do you create the observable stream (the behavior subject in this assignment)?

|  |
| --- |
| BehaviorSubject(self.note\_repository.get\_all\_notes()) |

Q19: How do you emit the new data (notes in this assignment) to the behavior subject?

|  |
| --- |
| on\_next(self.note\_repository.get\_all\_notes()) |

Q20: What is the role of the view model?

|  |
| --- |
| Subject that will be/being observed |

Q21: What are the main differences between the presenter and the view model?

|  |
| --- |
| The presenter updates the view instead of the view update itself but the view model let the view update itself automatically by observing it. |

Q22: In terms of testability, what do you think is easier to test between Presenter and ViewModel? Why?

|  |
| --- |
| ViewModel because when testing you would not have to create a mock view to test just testing on the viewModel layer but, to test the Presennter you have to create a mock view model to successfully test the Presenter. |

Next, we will move into the view implementation. Open main\_view.py and implement all the missing things.

**Hint 1:** The view will update itself by subscribing to the observable stream in the view model.

**Hint 2:** You will be mostly interacting with the view model.

Q23: How did you interact with the view model?

|  |
| --- |
| By subscribing it, view\_model.notes\_field.subscribe(lambda notes: self.update\_view(notes)) |

Q24: Put the screenshot of the MVVM Note Application displaying your members’ name in each note.

|  |
| --- |
|  |

Q25: What are pros and cons of MVVM pattern compared to MVC pattern?

|  |
| --- |
| Pros: Business logic is decoupled from Ul. Easy to maintain and test. Easy to reuse components  Cons: Hard to maintenance. Not offer tight coupling between the view and view model. |

Q26: According to MVC, MVP and MVVM pattern, what pattern would you prefer for your application? Why?

|  |
| --- |
| MVC  Because our project is not that large, and we don’t have much time to develop it. So we prefer mvc for easier development. |