Assignment 3: MVC, MVP and MVVM Architectural Pattern

Team member information:

Team Name: Prayath
Member 1: Nonthanat Theeratanapartkul (6031019821)
Member 2: Krit Kruaykitanon (6031002021)
Member 3: Nithipud Tunticharoenviwat (6031032921)
Member 4: Tanawit Kritwongwiman (6031021021)

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?

Design Principle for separation program into distinct parts so that each part will deliver separate concern. For example, the business layer contains the functions which drive an application’s core capabilities. On the other hand, the data layer comprises a database or other storage systems.

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

Business logic layers shouldn't be accessed directly from presentation layers. First, presentation layers often change. Presentation layers are implemented in order to be the only interface between users and the system. It controls the flow of users which should be separated from the business logic layers where all transactions are processed. If users can interact with business logic layers from presentation layers directly, someone could control your business logic process.

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
```

2. 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.

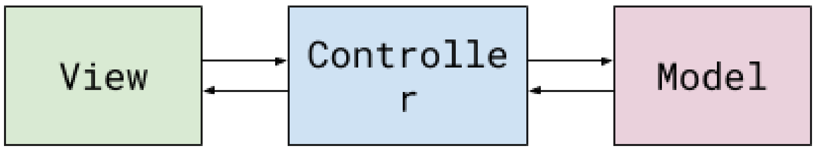
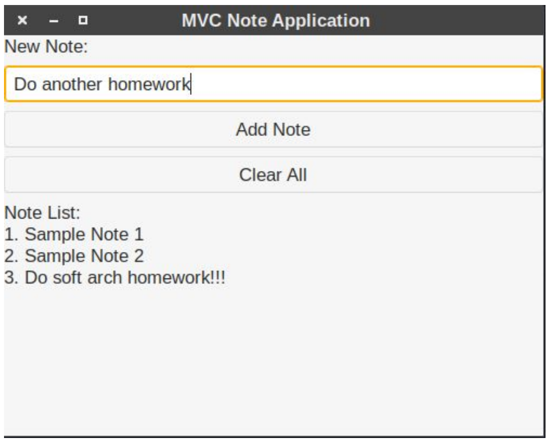


Figure: The MVC pattern and the interaction between each layer

We will create the simple note taking application as shown below


MVC Note Application

New Note:

Do another homework

Add Note

Clear All

Note List:
1. Sample Note 1
2. Sample Note 2
3. Do soft arch homework!!!

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?

The node repository was initialized, and node repository methods were implemented in order to make the main controller calls node repository methods properly.

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?

Originally, the UI was well implemented except some methods comprise a main controller. Next, these methods were added with functions that call the main controller's method. Lastly, update_view was added to the last line of these main views in order to refresh UI getting latest data.

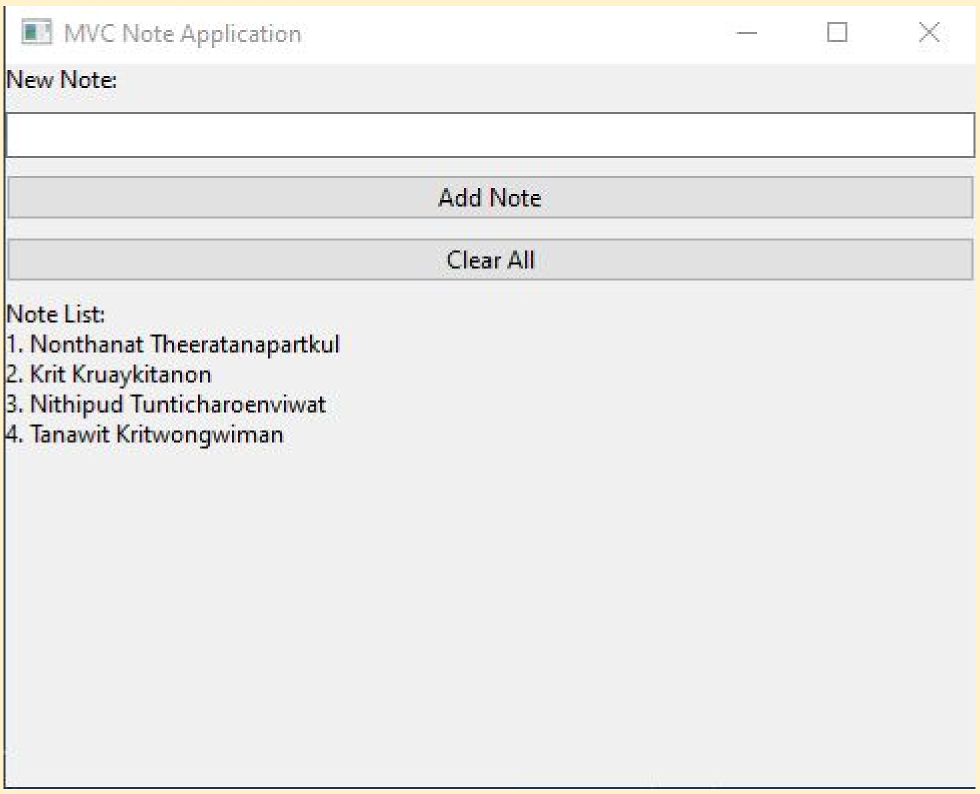
Q5: What is the role of the controller here? Explain it briefly.

It consists of business logics which are responsible for controlling the flow of program execution.
In this activity, the controller has its own note repository where all methods in it were implemented in order to function the repository. Also, the controller was initialized on the main view where inputs from UI decide it processes.

Q6: What are the advantages of MVC pattern?

Faster development + provide less duplication of logics + modification on view does not affect the entire model

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



Note List:
1. Nonthanat Theeratanapartkul
2. Krit Kruaykitanon
3. Nithipud Tunticharoenviwat
4. Tanawit Kritwongwiman

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.**

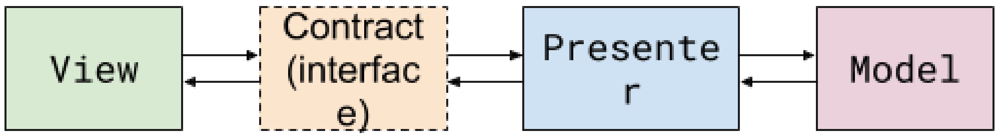


Figure: The MVP pattern and the interaction between each layer

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?

It helps reduce coupling between them.

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?

The presenter holds references to both the view and the model and is responsible for creating them and mediating between them. When the presenter holds the view reference, it would make the view updated instead of itself.

The presenter has functionality to receive input from user via View, then process data via Model and return result back to the view (update view instead of itself update)

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?

update_view function is being used in MainPresenter instead of update_view in View in MVC pattern

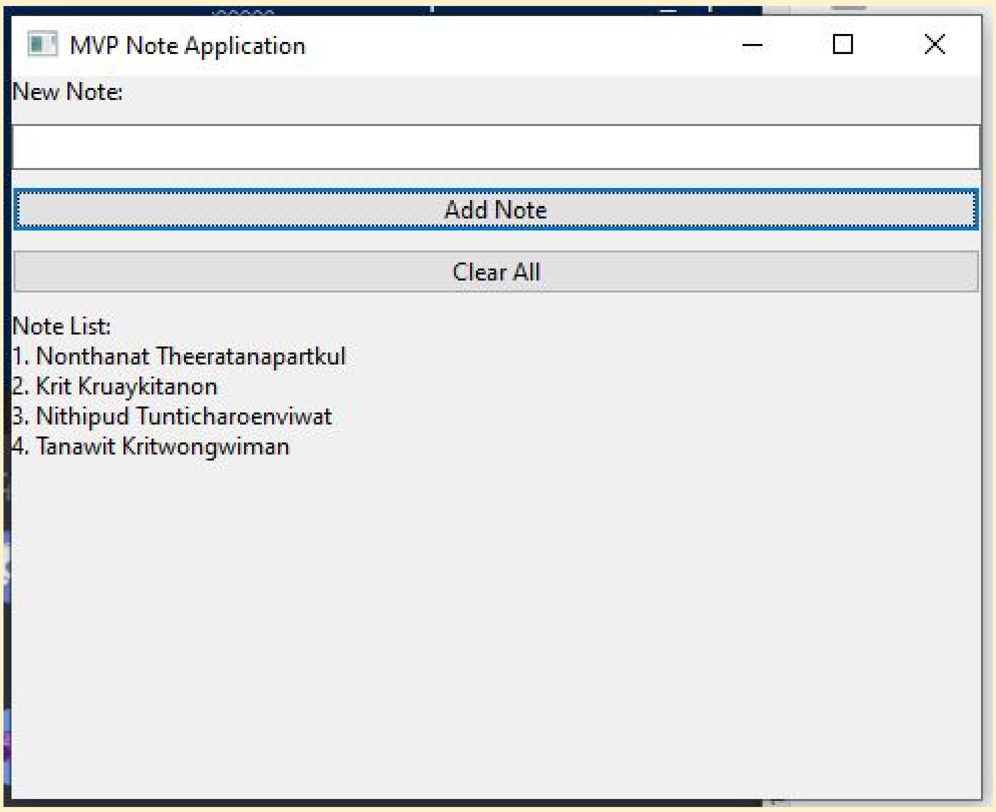
The next part is to implement the view so that it can interact with the presenter. If you look at the base_view.py, you will notice the set_presenter method which is called by the BasePresenter constructor. This way both view and presenter are now seeing each other.

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.

NOT SURE Q11: How did you interact with the presenter? Do you think it makes the implementation of view harder or easier? Why?

I did interact via call object presenter field in view object which is set in BasePresenter when initialize object **self.view.set_presenter(self)** which **set_presenter** is function defined in **BaseView** Class to set **self.presenter = self.presenter**
And I think it makes the implementation of view is more easier, since the business logic is moved into the presenter

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



Note List:
1. Nonthanat Theeratanapartkul
2. Krit Kruaykitanon
3. Nithipud Tunticharoenviwat
4. Tanawit Kritwongwiman

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

Even though it's decoupling view from controller via presenter and interface, when requirements grow bigger and bigger, the mvp style pattern still needs a lot of effort to implement and might owing to increasing the complexity of the code (Business logic increases in Presenter) and also resulting in more code.

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

Yes absolutely? With Well-defined interface? You would know actually what to test on each absolute separable component. As a result of separable logic, you could make unit tests or mock views easier.

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?

It is programming in a way to be concerned with data streams and the propagation of change.

Q16: What is the observer pattern?

It's one of the design patterns which object which here being called **subject**
This subject has the **observers** subscribing to it, and the subject could **notify** the observer automatically whenever the state of object 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.

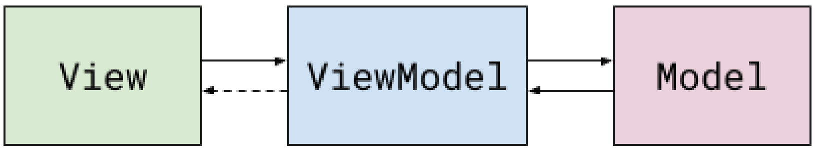


Figure: The MVVM pattern and the interaction between each layer

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?

It does not need to. Normally, view is the only component that needs to know what data is changed. Making viewmodel non-view-referencing, it's quite convenient to make viewmodel to be used on many views.

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)?

We create it via BehaviorSubject which has capability to represent value overtime and observers can subscribe to it to receive the last updated value.

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

Via **on_next** function

Q20: What is the role of the view model?

ViewModel is the processing of business logic via the subject.
When the user feeds input into view, the viewmodel notices via view calling method on the vm parameter referenced in view.
After they do process business logic handling with the use of Model, viewmodel could notify observers when view is needed changed.

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

The presenter does need to update the view itself. View model doesn't. Instead, they make a view as an observer and automatically update when data changes.

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

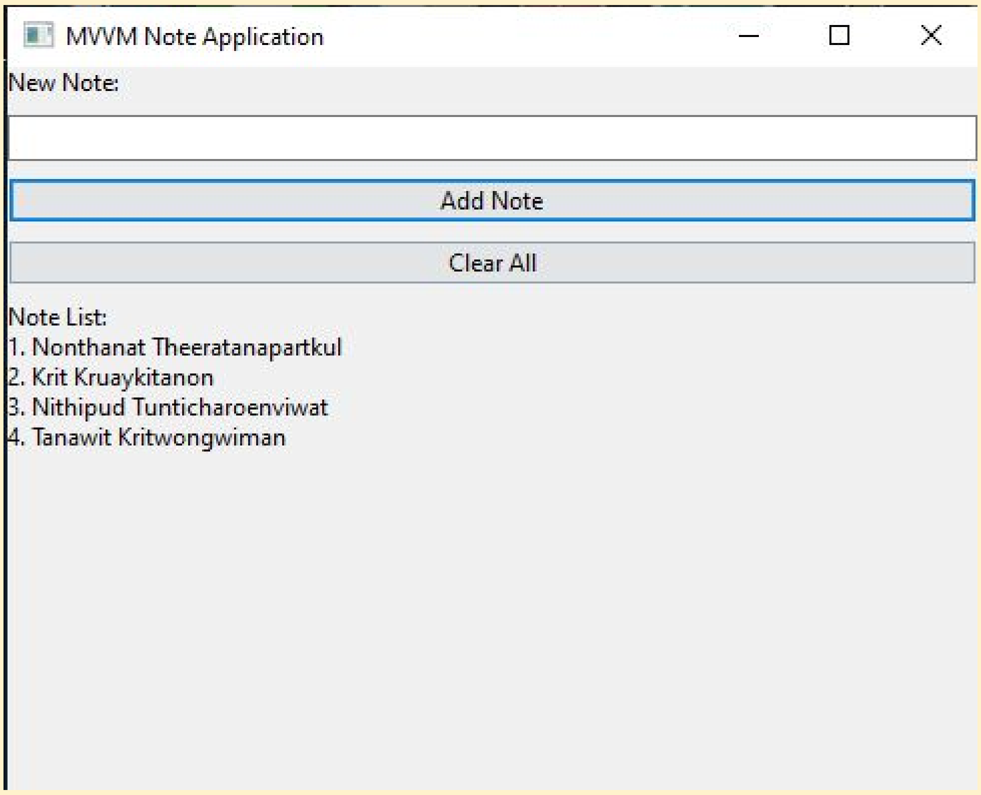
ViewModel since there is no need view reference

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?

We interact with the viewmodel via a directly calling viewmodel method with passing input data. To update the ui, we make subscribe to viewmodel object and order it to make **view_update** function

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



Note List:
1. Nonthanat Theeratanapartkul
2. Krit Kruaykitanon
3. Nithipud Tunticharoenviwat
4. Tanawit Kritwongwiman

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

Business logic is decoupled from UI in view. You could reuse the viewmodel component here exploiting the views sharing some of the same business logic.And as a result of multiple viewmodel to be used in many views, you could easily test and maintain the code more easily. Moreover, you could write a unit test case for both viewmodel and model without referencing the view. But there are still cons which are

1. lot of code maintenance are still hard to maintain
2. Tight coupling would make view model suffer

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

We choose MVC because our project isn't much large and we want to make project is easily to develop.