Assignment 3: MVC, MVP and MVVM Architectural Pattern

Team member information:

Team Name: TBD
Member 1: Keerati Chuatanapinyo (6030053921)
Member 2: Siwat Pongpanit (6030559121)
Member 3: Anawat Trongwattananon (6030631621)
Member 4: Thanapun Yan-amporn (6031022621)
Member 5: Weerayut Thinchamlong (6031055321)
Member 6: Setthanan Nakaphan (6031059921)

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?

Separation of concerns is a design principle that is concerned with separating programs which affect each other, View and Model, in this case.

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

No since if presentation layers were to be changed, then we have to implement new codes in the business logic layers for them to work with altered presentation layers.

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.

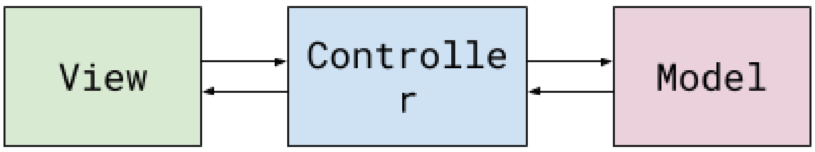
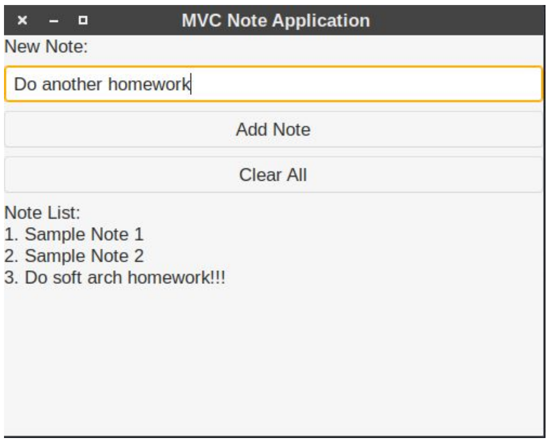


Figure: The MVC pattern and the interaction between each layer

We will create the simple note taking application as shown below



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?

#class MainController (main_controller.py)

```
    def __init__(self):
        # Create note repository here
        # Your code here
        self.repository = NoteRepository()

    def get_all_notes(self):
        # Return all notes
        # Your code here
        return self.repository.get_all_notes()

    def add_note(self, note: str):
        # Add note
        # Your code here
        self.repository.add_note(note)

    def clear_all(self):
        # Clear all note
        # Your code here
        self.repository.clear_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?

# class MainView(main_view.py)

```
    def on_clear_all_button_clicked(self, e):
        # Clear all note
        # Your code here
        # Update view
        # Your code here
        self.controller.clear_all()
        self.update_view(self.controller.get_all_notes())

    def on_add_note_button_clicked(self, e):
        content = self.note_input.GetValue()
        self.note_input.SetValue("")
        # Add new note
        # Your code here
        # Update view
        # Your code here
        self.controller.add_note(content)
        self.update_view(self.controller.get_all_notes())
```

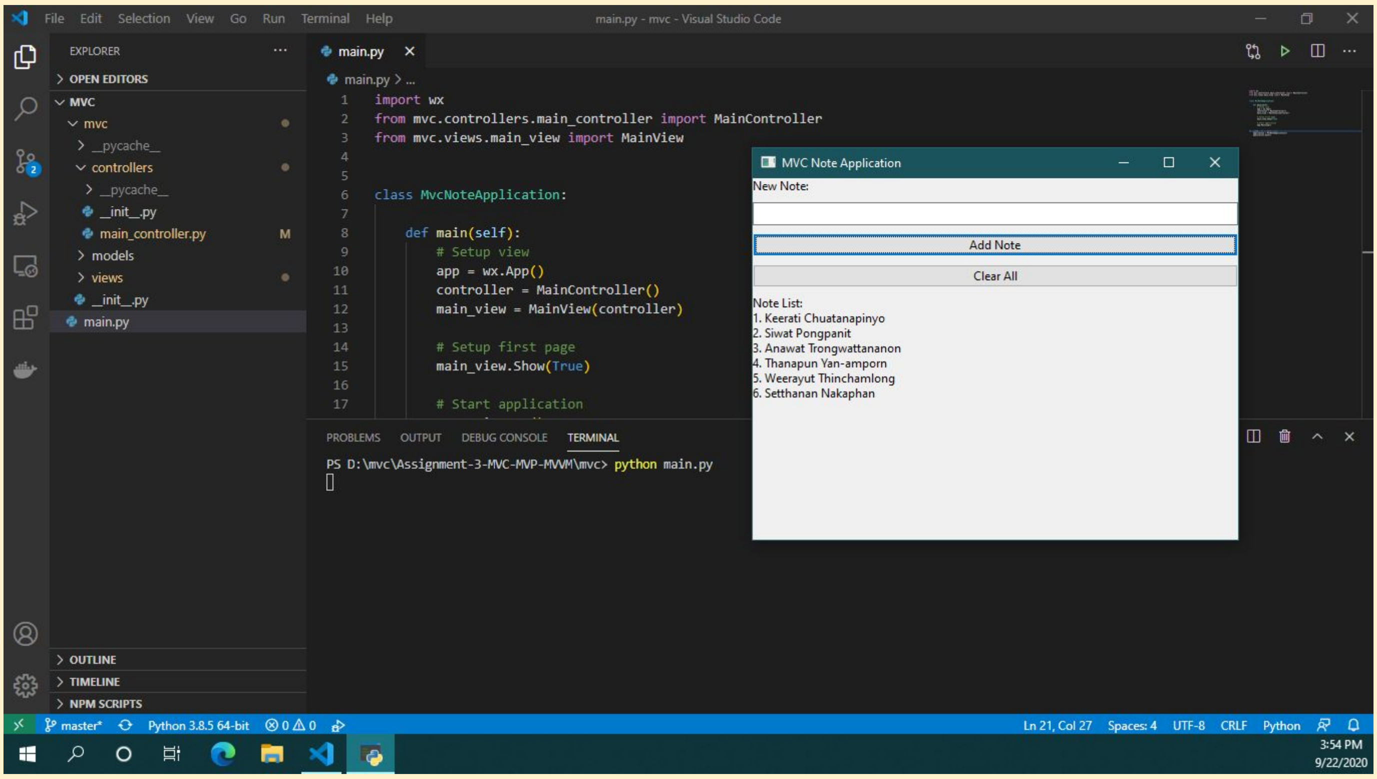
Q5: What is the role of the controller here? Explain it briefly.

Accepts input and converts it to commands for the model or view.

Q6: What are the advantages of MVC pattern?

- The modification does not affect the entire model
- Easier to Debug as we have multiple levels properly written in the application.
- Easy for multiple developers to collaborate and work together.
- Models can have multiple views

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



```
import wx
from mvc.controllers.main_controller import MainController
from mvc.views.main_view import MainView


class MvcNoteApplication:

    def main(self):
        # Setup view
        app = wx.App()
        controller = MainController()
        main_view = MainView(controller)

        # Setup first page
        main_view.Show(True)

        # Start application
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\mvc\Assignment-3-MVC-MVP-MVVM\mvc> python main.py
```

MVC Note Application

New Note:

Add Note

Clear All

Note List:
1. Keerati Chuatanapinyo
2. Siwat Pongpanit
3. Anawat Trongwattananon
4. Thanapun Yan-amporn
5. Weerayut Thinchamlong
6. Setthanan Nakaphan

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.

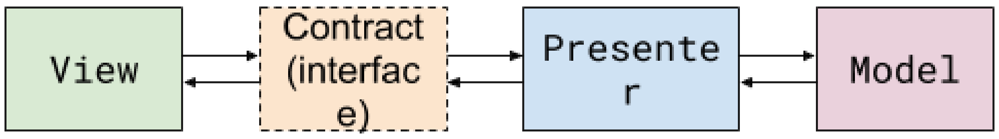


Figure: The MVP pattern and the interaction between each layer

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 is needed to be able to write an unit test replacing the implementation view with a test double.

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, a medium between View and Model, receives actions from View and changes caused by received actions from Model to update View.

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?

When the presenter was initialized it was bound to the view. Additionally, the view was updated in the presenter.

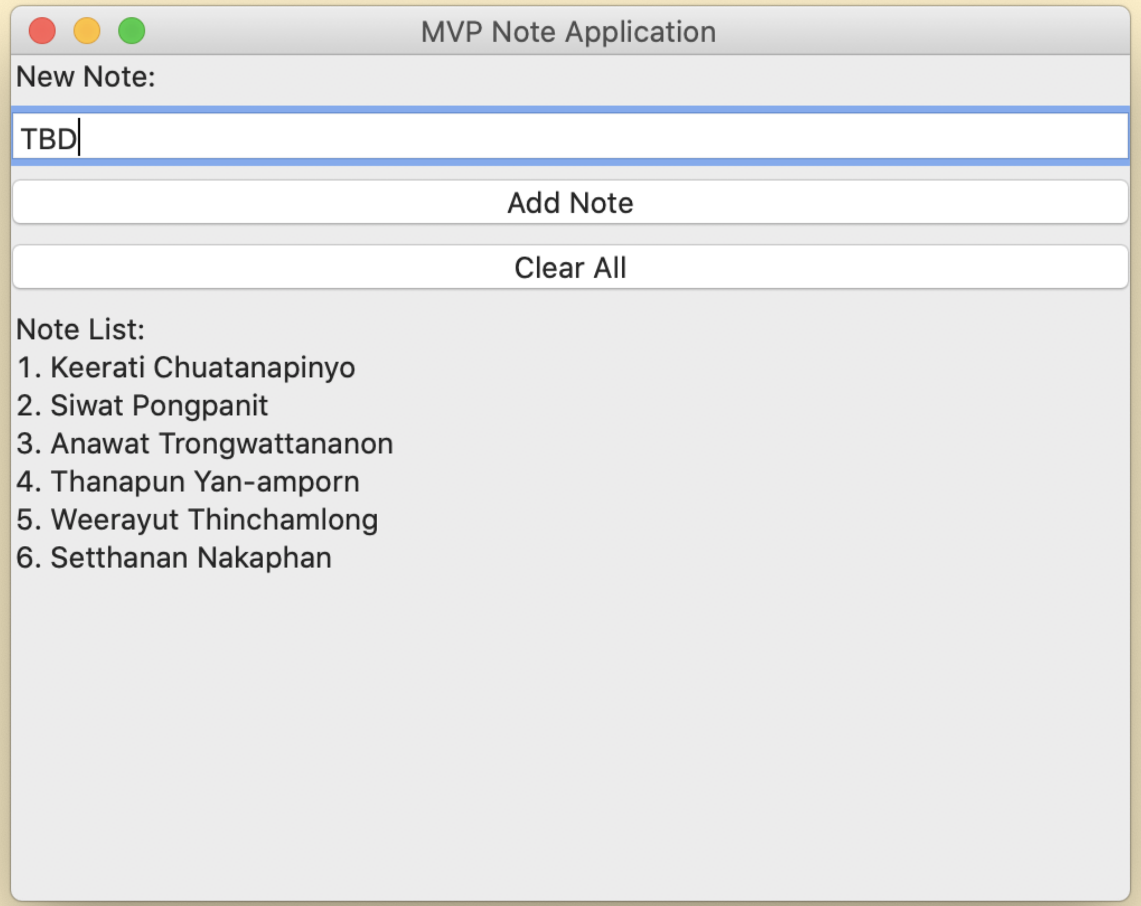
The next part is to implement the view so that it can interact with the presenter. If you look at the base_view.py, you will notice the set_presenter method which is called by the BasePresenter constructor. This way both view and presenter are now seeing each other.

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?

Through a reference that was assigned and initialized for the view when it was initialized.
I think it makes the implementation of the view slightly harder because you have to appoint a reference of the presenter and check if the reference exists.

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



Note List:
1. Keerati Chuatanapinyo
2. Siwat Pongpanit
3. Anawat Trongwattananon
4. Thanapun Yan-amporn
5. Weerayut Thinchamlong
6. Setthanan Nakaphan

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

Pros : More testable than MVC pattern
Cons : Number of codes, boiler-plate code, when the application is complex is more than MVC pattern.

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

Yes since we can test the view, the presenter, and the model in an isolated environment. For example, to test the view we just mock a presenter to give a fake data back to the view for it to render in any condition.

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 a program that combines function programming and observer together in order to make responses to the change that we are interested in.

Q16: What is the observer pattern?

The observer pattern is a design pattern in which an observable object notifies its observers automatically when there is a change in state of data that the observable is handling.

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.

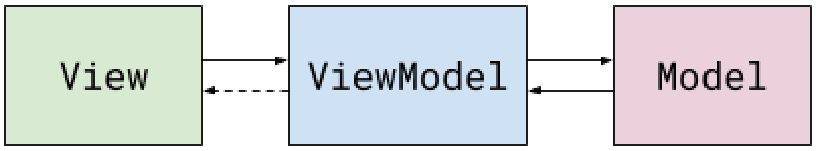


Figure: The MVVM pattern and the interaction between each layer

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?

No, because ViewModel is not responsible for updating the View, which it can observe changes and update itself if there are changes.

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)?

Create BehaviorSubject in main_view_model.py
code:
```
    self.note_behavior_subject = BehaviorSubject(self.note_respository.get_all_notes())
```

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

Add on_next to all functions in main_view_model.py
code:
```
    self.note_behavior_subject.on_next(self.note_repository.get_all_notes())
```

Q20: What is the role of the view model?

The view model acts as a medium in sending and receiving data to the view every time there are function calls.

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

Unlike Presenter, ViewModel is not required to reference a View.

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

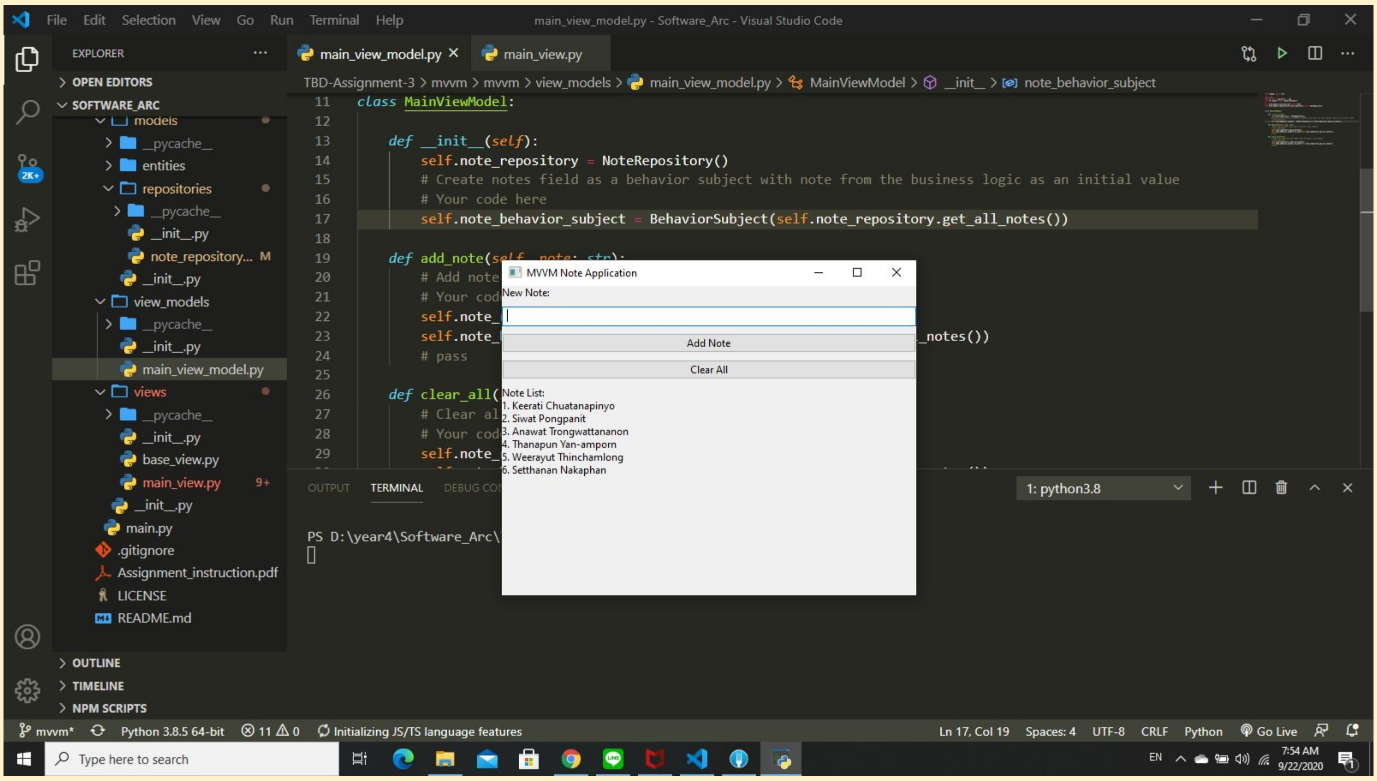
It is easier to test ViewModel because we can test it without mocking View like Presenter.

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?

Add subscribe to bind_observable in main_view.py
code:
```
    self.view_model.note_behavior_subject.subscribe( lambda note: self.update_view(note) )
```

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


main_view_model.py - Software_Arc - Visual Studio Code
main_view_model.py
main_view.py
TBD-Assignment-3 > mvvm > mvvm > view_models > main_view_model.py > MainViewModel > __init__ > note_behavior_subject
MVVM Note Application
Add Note
Clear All
1: python3.8

```
class MainViewModel:

    def __init__(self):
        self.note_repository = NoteRepository()
        # Create notes field as a behavior subject with note from the business logic as an initial value
        # Your code here
        self.note_behavior_subject = BehaviorSubject(self.note_repository.get_all_notes())

    def add_note(self, note: str):
        # Add note
        # Your cod
        self.note_
        self.note_                                          _notes())
        # pass

    def clear_all(
        # Clear al
        # Your cod
        self.note_
```

```
New Note:

Note List:
1. Keerati Chuatanapinyo
2. Siwat Pongpanit
3. Anawat Trongwattananon
4. Thanapun Yan-amporn
5. Weerayut Thinchamlong
6. Setthanan Nakaphan
```

```
PS D:\year4\Software_Arc\
```

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

Pros: More testable than MVC
Cons: Performance cost is more than in MVC pattern.

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

MVC since we will implement a web application, whose frontend part will be DOM manipulation and EventHandling, using javascript mainly.