Assignment 3: MVC, MVP and MVVM Architectural Pattern

Team member information:

Team Name: TTT-PY
Member 1: Natcha Manasuntorn 6030177021
Member 2: Karnkitti Kittikamron 6031006621
Member 3: Natthanon Manop 6031013021
Member 4: Suchut Sapsathien 6030609921
Member 5: Yanika Dontong 6031010021

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 (SoC) is a design principle to divide a program into distinct sections which each section focuses on a separate concern. A concern is a set of information that affects the code. SoC results in more flexibility for some aspect of the program's design, deployment, or usage. Well-separated concerns can lead to more module upgradability, reusability, and independent development.

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

No, we should not. Because direct access is one of the causes of high coupling between components in these layers which may lead to high complexity and low maintainability and flexibility. When the business logic layers changed, the presentation layers must be changed.

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.

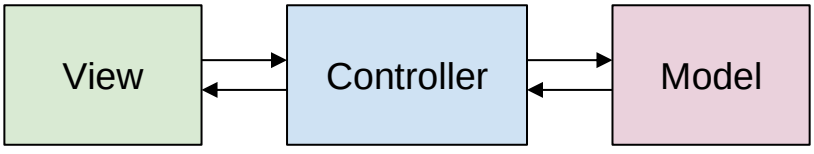


Figure: The MVC pattern and the interaction between each layer

We will create the simple note taking application as shown below

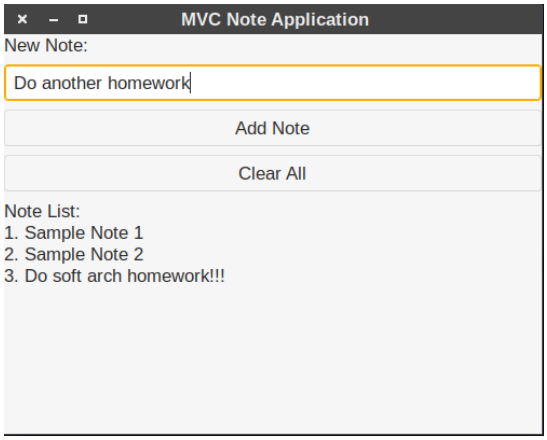


Figure: The simple MVC Note Application which we are going to create

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?

First, create NoteRepository from NoteRepository class in model. Then make each method in MainController class call a related method of the NoteRepository object. For example, make get_all_notes method in MainController class call get_all_notes method of NoteRepository object.

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?

There are 2 methods to be implemented to make view work: on_clear_all_button_clicked and on_add_note_button_clicked. In each method, we must implement 2 sections, one is for managing data and another one is update view. First, we call the controller method to manage data (add_note for on_add_note_button_clicked and clear_all for on_clear_all_button_clicked) then call update_view for refreshing view by receiving all notes from the controller (get_all_notes).

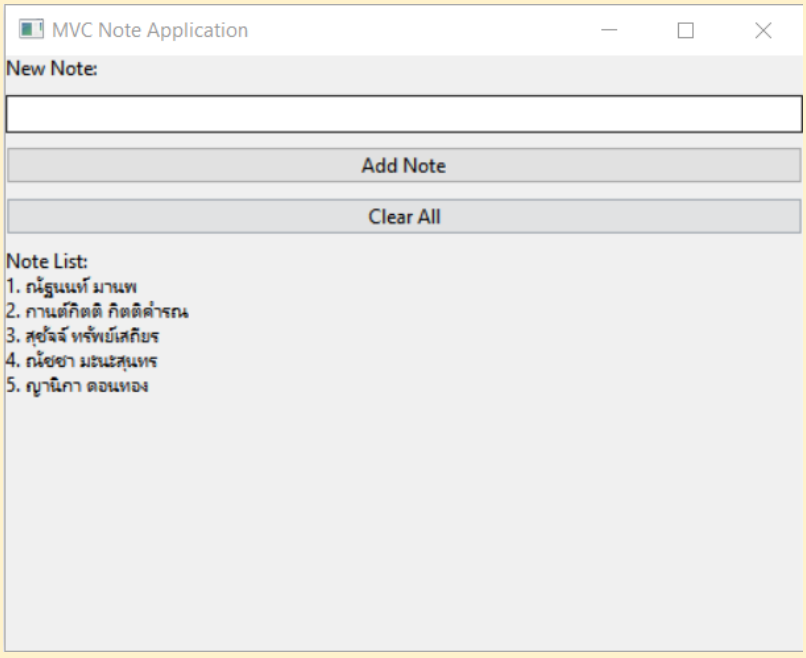
Q5: What is the role of the controller here? Explain it breifly.

Controller is the class which is represented as a middleman between view and model. The view communicates with the controller for data management then the controller will communicate with the model to do the logic behind for changing data entities.

Q6: What are the advantages of MVC pattern?

1. Ease of understanding
2. Independent development
3. Reusability

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.

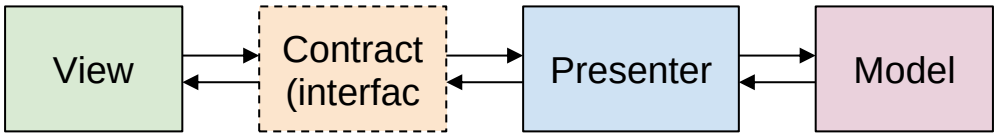


Figure: The MVP pattern and the interaction between each layer

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?

Introducing an interface between the view and the presenter supports developers to implement the layers correctly by identifying methods that need to be implemented from each other's side.

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 plays a role in controlling business logic and 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?

The main difference is that MainController is only responsible for controlling business logic, but MainPresenter is responsible for both controlling business logic and updating view.

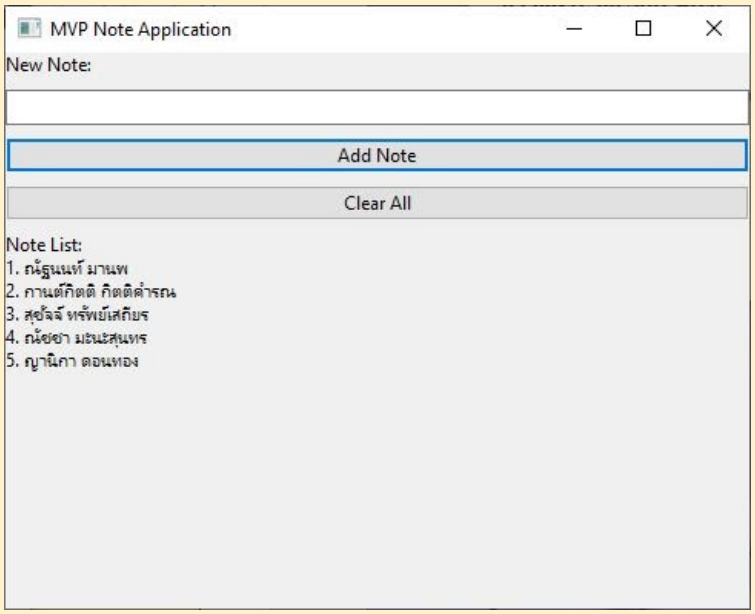
The next part is to implement the view so that it can interact with the presenter. If you look at the base_view.py, you will notice the set_presenter method which is called by the BasePresenter constructor. This way both view and presenter are now seeing each other.

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?

View interact with presenter by calling function through contract(interface) with this implementation of view, it will be easier when some function that appear in every class are needed to be changed and it was easier when writing a test.

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?

	Pro	**Con**
MVP	• Easier to testing • Concern is separated • Easier to Maintain	• Difficult to understand because presenter is prone to adding additional business logic • Boilerplate code
MVC	• Concern is separated	• Hard to unit test it because controller is tried to framework • Tight coupling between controller and views • Take over time to maintenance because code is transferred to controllers which make in bloated

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

Yes, it is because MVP focuses on decoupling dependencies with the help of interfaces which simplifies unit testing. Meanwhile, MVC is no separate component to handle UI or Presentation logic. the code is written in View layer which is more difficult.

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?

A declarative programming paradigm concerned with data streams and the propagation of change

Q16: What is the observer pattern?

A software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

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.

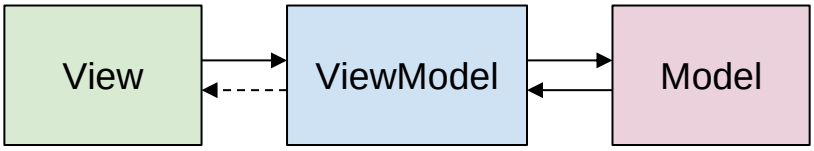


Figure: The MVVM pattern and the interaction between each layer

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 MVVM has a binder, which automates communication between the view and its bound properties in the view model. ViewModel does not hold the reference to the View.

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)?

```
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())
```

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

```
def add_note(self, note: str):
    # Add note and emit event with new date to the subject
    # Your code here
    self.note_repository.add_note(note)
    self.note_behavior_subject.on_next(self.note_repository.get_all_notes())
    pass
```

Q20: What is the role of the view model?

A value converter, meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented.

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

ViewModel does not hold the reference to the View but presenter is to drive the View changes and at the same time provide data for completing those operations. ViewModel only provides the data, whereas the View is responsible for consuming them.

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

MVVM is easier because MVVM view has no need to be mocked for testing unlike MVP

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?

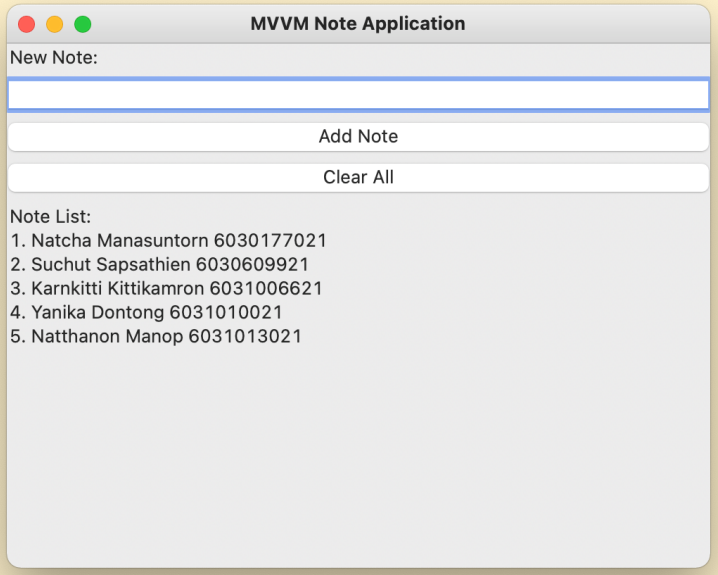
```
def __init__(self):
    BaseView.__init__(self, "MVVM Note Application")
    # Create view model
    # Your code here
    self.view_model = MainViewModel()

    self.init_ui()

    # Bind observable
    self.bind_observable()

def bind_observable(self):
    # Subscribe to the notes behavior subject and update the view when the data change
    # You  view_model: MainViewModel
    self.view_model.note_behavior_subject.subscribe( lambda updated_note : self.update_view(updated_note) )
    pass
```

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?

	Pro	**Con**
MVC	• Easier support for a new type of clients • All classed and objects are independent of each other so that you can test them separately • MVC allows logical grouping of related actions on a controller together.	• Business logic is mixed with UI • Increased complexity and Inefficiency of data • Hard to reuse and implement tests
MVVM	• Concern is separated • Less boilerplate codes • View components are easy to reuse • Easy to reuse components	• Hard to control view • hard to maintain code

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

Since our application is chat system based on web application, MVC would be the most suitable pattern. There are 3 reasons why we choose MVC pattern.
1. The application can be divided into small units to avoid complexity.
2. The development of the various components can be performed concurrently.
3. The views and controllers are pluggable.