# A Clean Architecture for a PyQT GUI Using the MVP Pattern

Markus Huber Follow 6 min read · Sep 20, 2023



Building a GUI can quickly become a daunting task, especially when aiming for a scalable and maintainable design. In this article, we'll explore how to create a GUI that collects user input and prints it out upon pressing a button, all while adhering to a clean architecture using the Model-View-Presenter (MVP) pattern.

## Introduction

In the world of software, Graphical User Interfaces (GUIs) are crucial for delivering a seamless user experience. However, as these applications grow in complexity, so does the challenge of maintaining and scaling them. The Model-View-Presenter (MVP) pattern emerges as a beacon, offering a structured approach to designing organized and maintainable GUIs. In this article, we'll unravel the MVP pattern and apply it to a simple project: a GUI that captures user input and displays it upon a button press. Let's explore how to keep our GUI applications both user-friendly and architecturally sound.

## Understanding the MVP Pattern

The MVP pattern is a derivative of the well-known Model-View-Controller (MVC) architecture. While both architectures aim to separate concerns in an application, MVP is particularly tailored for modern GUI applications, providing a more defined separation between the View and the Model.

### 1. Model

- Role: The Model is responsible for the application's data and its business logic. This includes data retrieval, storage, and processing.
- Features: It is completely unaware of the UI's existence. Any change in the UI won't affect the Model, ensuring that business logic remains consistent irrespective of UI changes.
- Example: In our GUI application, the Model would store the user's input and provide methods to retrieve or update this data.

### 2. View

- Role: The View is, in essence, the GUI of the application. It displays data to the user and sends user commands to the Presenter.
- Features: The View is passive, meaning it doesn't have any knowledge about the business logic. It simply displays what the Presenter tells it to and informs the Presenter about user interactions.
- Example: In our context, the View consists of input fields for the user and a "start" button. When the button is clicked, the View notifies the Presenter.

### 3. Presenter

- Role: The Presenter is the mediator that decouples the Model and the View. It retrieves data from the Model, manipulates it, and sends it to the

View for display. Conversely, it also takes user inputs from the View, processes it (if necessary), and updates the Model.

- Features: Unlike a controller in MVC, the Presenter is more involved in the UI logic. However, it doesn't have a direct reference to the UI framework, making it easier to unit test.
- Example: When our GUI's "start" button is pressed, the Presenter fetches data from the Model, processes it, and might update the View or Model based on the application's logic.

### Benefits of MVP:

- Decoupled Architecture: Each component (Model, View, Presenter) operates independently, which means changes in one component don't directly affect the others. This separation of concerns is crucial for maintainability.
- Testability: Since the business logic is separated from the UI and the Presenter doesn't have a direct reference to the UI framework, it becomes much easier to write unit tests for the business logic.
- Flexibility: The clear boundaries between components mean that, for instance, the UI can be overhauled without having to touch the business logic. Similarly, business logic can be modified without affecting the UI.

## MVC vs MVP

In UI design, both MVC (Model-View-Controller) and MVP (Model-View-Presenter) aim to separate data, presentation, and user input. In MVC, the user interacts with the View, which passes commands to the Controller, and then the Model updates the View. In MVP, the View sends user interactions to the Presenter, which handles communication with the Model. The Model's feedback to the View is channeled through the Presenter. This separation in

MVP aligns well with PyQt's signal-slot mechanism. This compatibility, along with better modularity and testability, drove our choice towards MVP for the GUI.

## Setting Up the Project

Folder structure:

```
mvp/
  model.py
  view.py
  presenter.py
ui/
  mainwindow_ui.py
main.py
```

## Dive into the Code

### The Model ( `mvp/model.py` )

The Model serves as the backbone of our application, handling data management and any business logic. It ensures that our application's core functionalities remain independent of the user interface (View). This abstraction allows for flexibility and scalability, ensuring that changes in the UI won't directly affect the underlying logic.

```
from PyQt6.QtCore import QObject
from typing import Dict

class Model(QObject):
    def set_input_data(self, data_dict: Dict[str, str]) -> None:
        self.input_data = data_dict
```

```
    def get_input_data(self) -> Dict[str, str]:
        return self.input_data
```

In this example, the Model class provides methods to set and retrieve user input data, ensuring a neat separation from the View.

### The View ( `mvp/view.py` )

Represents the UI, acting as the direct interface between the user and the software. Within our design, the View primarily handles the graphical elements, capturing user interactions, and relaying these events to the Presenter. It contains user input fields where users can input their data, and a “start” button to trigger the process of gathering and displaying the entered data.

Example:

```
from PyQt6.QtWidgets import QDialog
from PyQt6.QtCore import pyqtSignal
from typing import Dict

class MyDialog(QDialog, Ui_Dialog):
    input_data_collected = pyqtSignal(dict)

    def on_start_clicked(self) -> None:
        data_dict: Dict[str, str] = {
            'inputEdit1': self.lineEdit1.text(),
            # ... Add more input fields as needed ...
        }
        self.input_data_collected.emit(data_dict)
```

In this example, the `MyDialog` class, representing the View, captures user inputs from multiple fields and emits them as a dictionary when the "start"

button is clicked. (The code is missing the initialization for the PyQT GUI and the start button connection to the `on_start_clicked` function. See working example in my repo)

### The Presenter (`mvp/presenter.py`)

The Presenter plays a pivotal role in the MVP pattern, ensuring that the Model and View remain decoupled. By acting as an intermediary, it handles user interactions from the View, processes the data if necessary, and then updates the Model. This structure ensures a clear separation of concerns, making the application more maintainable and scalable.

Example:

```
from typing import Dict

class Presenter:
    def __init__(self, model: Model, view: MyDialog) -> None:
        self.model = model
        self.view = view
        self.view.input_data_collected.connect(self.handle_input_data)

    def handle_input_data(self, data_dict: Dict[str, str]) -> None:
        self.model.set_input_data(data_dict)
        print(data_dict)
```

In this program example, the `Presenter` acts as an intermediary between the `Model` and `View`, listening for user input collected in the View, updating the Model with this data, and then printing it to the console.

## Bringing It All Together (`main.py`)

Creating an application using the MVP pattern requires careful orchestration of its three core components. The `main.py` script serves as the entry point for our application, where we tie the Model, View, and Presenter together, initializing them and setting the app in motion.

### Step 1: Import Necessary Modules

Before we start, we need to import the necessary modules:

```
import sys
from PyQt6.QtWidgets import QApplication
from mvp.model import Model
from mvp.view import MyDialog
from mvp.presenter import Presenter
```

### Step 2: Create the Application Instance

Every PyQt application begins with an instance of `QApplication`. This manages GUI control flow and main settings.

```
app = QApplication(sys.argv)
```

### Step 3: Instantiate Model, View, and Presenter

With the application instance in place, we can instantiate our MVP components. We'll start by creating an instance of our `Model` and `View`

```
model = Model()
view = MyDialog()
```

The `Presenter` is where the magic happens. When creating the `Presenter`, we pass in our `Model` and `View` instances, allowing it to mediate interactions between them.

```
presenter = Presenter(model, view)
```

### Step 4: Display the GUI

Once everything is set up, we can display our main dialog:

```
view.show()
```

### Step 5: Execute the Application's Main Loop

Finally, to start the application's main loop, we call:

```
sys.exit(app.exec())
```

This loop continuously checks for and dispatches events. When the loop exits, the application will close.

### Complete `main.py`:

Combining all the steps, here's what our `main.py` looks like:

```
import sys
from PyQt6.QtWidgets import QApplication
from mvp.model import Model
from mvp.view import MyDialog
from mvp.presenter import Presenter

def main():
    app = QApplication(sys.argv)

    model = Model()
    view = MyDialog()
    presenter = Presenter(model, view)  # This implicitly ties model and view to

    view.show()
    sys.exit(app.exec())
if __name__ == "__main__":
    main()
```

## Conclusion

The MVP pattern, while adding a layer of abstraction, greatly improves the organization of GUI applications. By separating concerns, it allows for more maintainable and scalable code. With `main.py` serving as the entry point, we have a clear and concise overview of how the application's components come together.

## References

See full working repo here.

Pyqt Python MVP Mvc

## Written by Markus Huber

19 followers · 13 following

Follow

My articles are written with the help of chatgpt

## Responses (1)

Aunuun Jeffry Mahbuubi

What are your thoughts?

Ed Eaglehouse
Feb 18, 2025 (edited)

You're describing MVC differently than the way it is actually used, at least in its pure form. The Model is a passive class used only for holding data and metadata about what constitutes valid values - it knows nothing about the View or Controller... more

Reply

## More from Markus Huber


Markus Huber

## Decoding the JWT Anomaly: When Changing a Token's Last Characte...

JWT (JSON Web Token) is widely used as a compact and URL-safe means of representin...

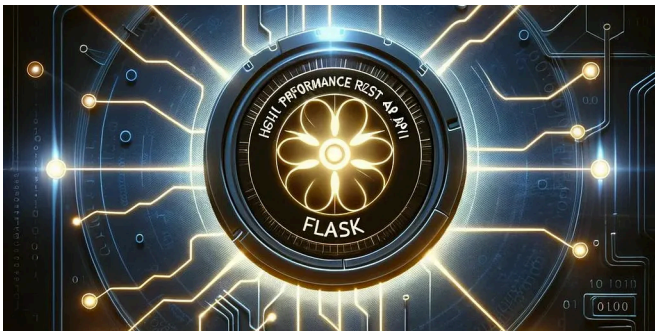
Oct 21, 2023 4

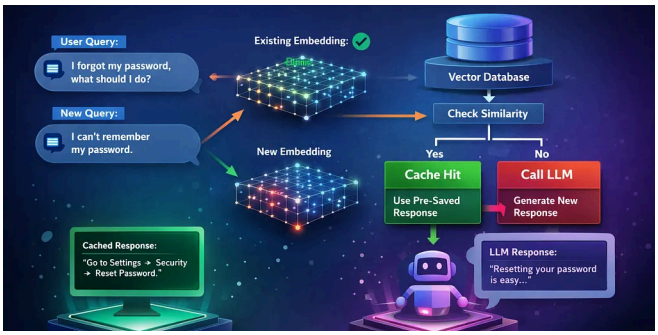
Markus Huber

## When a Redirect Takes the Wrong Turn: Demystifying...

Have you ever encountered the enigmatic "NEXT_REDIRECT" message lurking in your...

Jan 27, 2024 3 2




Markus Huber

## Building and Scaling High-Performance REST APIs: A Tutori...

This is the first part of a multi part series. Having a well-constructed API can...

Apr 28, 2024 3

Markus Huber

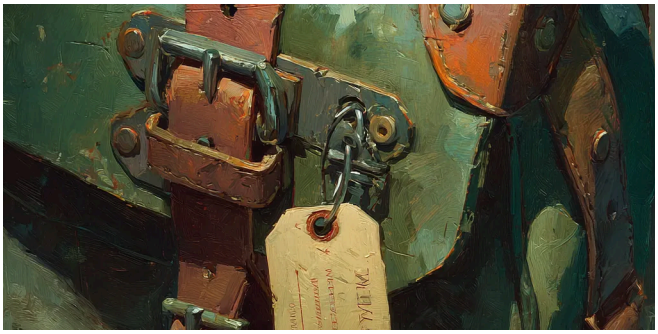
## Semantic Caching for LLMs (and Why It Feels Obvious in Hindsight)

If you've worked with LLMs in production, you've probably hit this moment:

Jan 1

See all from Markus Huber

# Recommended from Medium

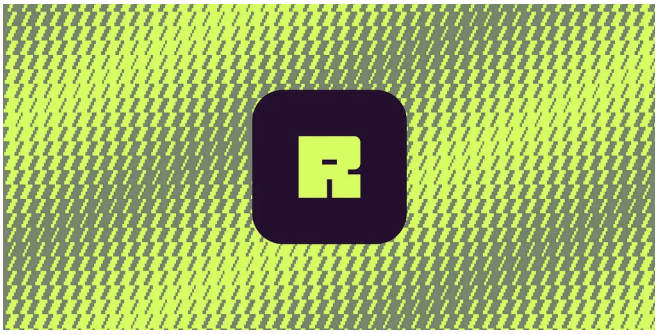


In Intuitively and Exhaustively Ex... by Daniel Warf...

## AI Generated In-Text Citations—Intuitively and Exhaustively...

Knowing where information comes from in LLM-powered applications

Sep 3, 2025 163

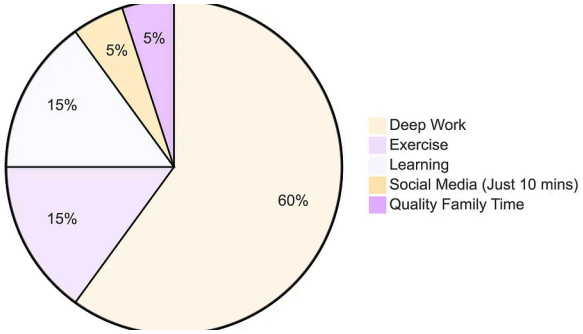


MA Raza, Ph.D.

## Ruff Tutorial: A Complete Guide for Python Developers

What is Ruff?

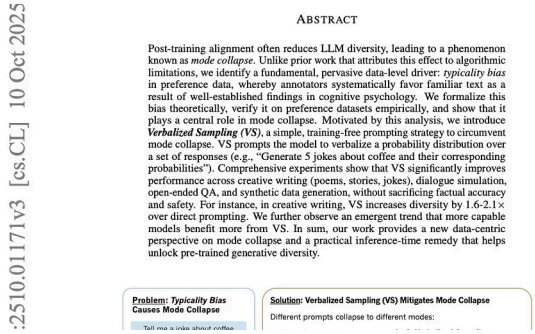
Aug 15, 2025 125 1




In Level Up Coding by Teja Kusireddy

## I Stopped Using ChatGPT for 30 Days. What Happened to My Brai...

91% of you will abandon 2026 resolutions by January 10th. Here's how to be in the 9% who...

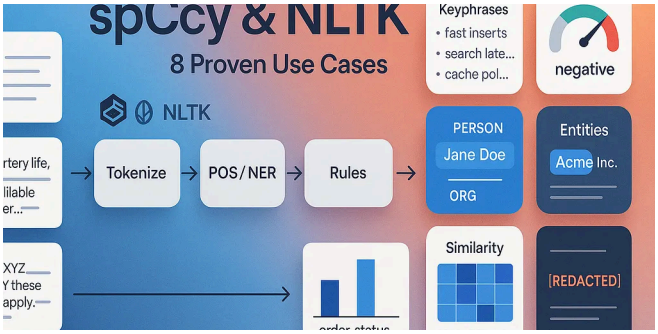



In Generative AI by Adham Khaled

## Stanford Just Killed Prompt Engineering With 8 Words (And I...

ChatGPT keeps giving you the same boring response? This new technique unlocks 2×...

Dec 29, 2025 3.7K 153




Nexumo

## 8 spaCy/NLTK Use Cases That Still Matter (with Code)

Practical NLP tasks where classic libraries beat “just use a giant LLM,” with short,...

Oct 8, 2025 109 2

Oct 20, 2025 23K 608

In Python in Plain English by Brent Fischer

## Understanding Web Authentication: A Practical Guide...

Authentication is one of the most misunderstood areas in web development....

Dec 18, 2025 1

See more recommendations