**Project Report**

**Weather App**

**Fall 2024**

**Object Oreinted Programming**

Submitted by: **Faizan Muhammad**

Registration No. : **20PWCSE1972**

Class Section: **Repeater**

“On my honor, as students of University of Engineering and Technology, We have neither given nor received unauthorized assistance on this academic work.”

Submitted to:

**Summaya Salahuddin**

DATED: 28 January 2025 , 2025

**Department of Computer Systems Engineering**

**University of Engineering and Technology, Peshawar**

Contents

[**INTRODUCTION** 3](#_Toc188933493)

[**OOP CONCEPTS APPLIED** 5](#_Toc188933494)

[**Project Structure** 6](#_Toc188933495)

[**Algorithm and Logic** 7](#_Toc188933496)

[**FLOWCHART** 10](#_Toc188933497)

[**OOP in Action: Detailed Class Explanation** 11](#_Toc188933498)

[**Future Implementation:** 15](#_Toc188933499)

[**CONCLUSION:** 16](#_Toc188933500)

# **INTRODUCTION**

**Overview of the Weather App Project**

The Weather App project is a Python-based graphical user interface (GUI) application designed to provide real-time weather updates to users. It allows users to input the name of any city, fetch weather data, and view the current temperature, weather description, and a weather-related emoji (such as sunny, rainy, or snowy). The app uses the OpenWeatherMap API to gather weather data, which is then displayed in an intuitive and visually appealing format using PyQt5 for the front-end GUI development.

This project demonstrates the ability to combine **API integration** with **GUI design**, offering a real-time experience that fetches data from an external source (OpenWeatherMap) and presents it in an accessible and interactive format.

**Purpose**

The main goal of this project is to create a simple but fully functional weather application where users can quickly and easily check the weather for any city in the world. The app is designed to be user-friendly and interactive, allowing users to input the name of a city and instantly retrieve relevant weather information.

The app was built with the following objectives:

* **Interactive GUI**: To allow users to interact with the app via a simple input field for the city name and a button to fetch the weather information.
* **Real-Time Data Fetching**: To make API calls to the OpenWeatherMap service to fetch real-time weather data, ensuring the app provides accurate and up-to-date weather reports.
* **User-Friendly Display**: The weather data (temperature, weather conditions, etc.) is displayed clearly, with the added benefit of showing weather-related emojis to enhance the visual appeal and make the information more easily recognizable.
* **Error Handling**: The app also provides feedback in case of an error, such as invalid city input or API request failure, which helps users understand what went wrong.

**Technologies Used**

This project utilizes several technologies to ensure a smooth and responsive user experience:

* **Python**: The core programming language used to develop the entire app. Python’s simplicity and power make it an ideal choice for this project, allowing the creation of the app’s logic and integration with external APIs.
* **PyQt5**: This is the primary library used for developing the graphical user interface (GUI). PyQt5 provides various built-in widgets, layouts, and tools for creating desktop applications. In this project, PyQt5 was used to build the input form (city name entry), the buttons, and the display of weather data. The flexibility and ease of use of PyQt5 allowed for quick development of a modern and clean interface.
* **OpenWeatherMap API**: The OpenWeatherMap API is a free and widely used API that provides weather information globally. The app communicates with this API to fetch current weather data based on the user’s input (city name). The response from the API is then processed and displayed to the user in the app.

These technologies were chosen for their ease of use, reliability, and extensive community support, ensuring that the project would be both functional and easy to maintain.

**The Importance of Object-Oriented Programming (OOP)**

Object-Oriented Programming (OOP) played a central role in the design and development of this application. OOP is a programming paradigm that helps organize code into **objects** that represent real-world entities and actions. By applying OOP principles, we were able to create a **modular, maintainable, and scalable** application.

Here’s how OOP principles were beneficial in this project:

* **Encapsulation**: We encapsulated all the weather-related logic (fetching data from the API, processing it, displaying it) and UI interactions (button clicks, city name input) inside a single WeatherApp class. This helped in organizing the code logically and keeping the components self-contained. For example, all code related to displaying the weather is encapsulated in methods like display\_weather(), while the error handling is handled in the display\_error() method. This makes the code easier to understand and maintain.
* **Abstraction**: Abstraction allows us to hide the complexity of the weather data fetching process. The user doesn’t need to worry about the technical details of how the app fetches the data from the OpenWeatherMap API. Instead, they interact with simple methods like get\_weather() that abstract away the complexities and allow the user to focus on the input/output. This improves both user experience and code readability.
* **Inheritance**: The WeatherApp class inherits from PyQt5’s QWidget, which provides basic functionality for the window and widgets. Inheriting from QWidget allows the app to leverage the built-in PyQt5 functionalities, such as event handling and layout management, without needing to implement them from scratch.
* **Polymorphism**: Polymorphism in this project is reflected in how different widgets (like the labels, input fields, and buttons) can be manipulated using the same interface. For instance, all the widgets (labels and buttons) are styled and aligned similarly using uniform methods. The same method setAlignment() is applied to multiple widgets (with different types) to ensure a consistent UI, demonstrating the power of polymorphism in simplifying the interaction with different objects.

By adopting OOP principles, the app is structured in a way that makes it easy to modify, extend, and debug. These principles allow for **better code reuse** and **future scalability**, which is especially important for maintaining and enhancing the application over time.

# **OOP CONCEPTS APPLIED**

**OOP Concepts Applied**

In this project, several core Object-Oriented Programming (OOP) concepts were applied to ensure the code is modular, maintainable, and scalable. These concepts are essential for organizing and structuring the application efficiently. Below are the key OOP principles used in the Weather App:

1. **Encapsulation**:  
   The project uses encapsulation by grouping related functionality and data into the WeatherApp class. For example, all the methods responsible for handling user input, fetching data from the OpenWeatherMap API, and displaying the results are encapsulated within this class. This helps to **hide the internal workings** of the application, exposing only the necessary interactions to the user (such as clicking a button to get the weather), making the code easier to manage and debug.
2. **Abstraction**:  
   Abstraction allows the app to **simplify** complex operations. The logic of fetching weather data from the API is abstracted in the get\_weather() method, so the user only interacts with a simple GUI (city name input and a button) rather than having to understand how the API is being called or how the data is parsed. The user is shielded from unnecessary details, focusing only on the input/output process.
3. **Inheritance**:  
   The WeatherApp class inherits from PyQt5’s QWidget class, enabling it to take advantage of built-in functionality such as event handling, layout management, and widget creation. By inheriting from QWidget, the app can focus on implementing only the unique behaviors required for the weather app, without needing to reimplement common GUI components.
4. **Polymorphism**:  
   Polymorphism is utilized in how the program interacts with different types of widgets (e.g., labels and buttons) using a common interface. For example, the method setAlignment() is applied to various types of widgets (labels, input fields, etc.), demonstrating that polymorphism allows for consistent treatment of different objects in a uniform way, improving code reusability and readability.

These OOP principles ensure that the Weather App remains organized, easy to extend, and more adaptable to future modifications, such as adding new features or refactoring the UI without major disruptions to the core functionality.

# **Project Structure**

The structure of the Weather App project is designed to be modular, well-organized, and easy to maintain. By following object-oriented principles and focusing on logical separation of concerns, the project is divided into key components that manage distinct responsibilities. Here’s an overview of the project structure:

**1. Main Application (WeatherApp class)**

At the core of the project is the WeatherApp class, which encapsulates the entire functionality of the application. This class is responsible for initializing the graphical user interface (GUI), handling user interactions, and managing the communication between the user interface and the backend API.

Key methods in the WeatherApp class:

* **\_\_init\_\_()**: Initializes the GUI elements such as labels, buttons, and input fields. It sets up the layout and styling of the app.
* **initUI()**: Configures the layout and appearance of the widgets. It also aligns and styles them for a user-friendly interface.
* **get\_weather()**: Handles the logic for fetching weather data from the OpenWeatherMap API based on the user’s input (city name). This method processes the API response and passes the data to the display functions.
* **display\_error()**: Displays error messages to the user when there is an issue (e.g., invalid city name or failed API request).
* **display\_weather()**: Displays the fetched weather data (temperature, weather description, and emoji) in the appropriate labels on the GUI.
* **get\_weather\_emoji()**: Returns the corresponding emoji based on the weather condition ID, enhancing the visual representation of the weather.

**2. GUI Components**

The user interface is composed of the following key components:

* **QLabel**: Labels are used to display the city name prompt, weather information (temperature, description), and error messages.
* **QLineEdit**: A text input field where users can type the name of the city they want to check the weather for.
* **QPushButton**: A button that, when clicked, triggers the weather fetch operation.
* **QVBoxLayout**: A vertical layout manager that arranges the GUI components in a column, ensuring the interface is clean and properly aligned.

These components are grouped together logically in the layout, ensuring a smooth flow of interaction.

**3. External Dependencies**

The project relies on a few key external libraries:

* **PyQt5**: Provides the necessary tools for building the graphical user interface (GUI) components. This library includes various widgets, layout management options, and event handling features.
* **requests**: Used to send HTTP requests to the OpenWeatherMap API to fetch weather data. It handles the network communication between the application and the weather service.
* **OpenWeatherMap API**: A third-party service that provides weather data. The API returns information such as temperature, humidity, weather condition, and more, based on the city name.

**4. Event Handling**

Event handling is crucial in the Weather App project. The app listens for the **button click event** when the user presses the “Get Weather” button. When the button is clicked, the app triggers the get\_weather() method to fetch data from the OpenWeatherMap API and update the display accordingly.

**5. Error Handling**

Error handling is built into the get\_weather() method to ensure that if the API request fails or if the user provides invalid input, appropriate error messages are displayed. The display\_error() method ensures that the error message is clear and visible, enhancing the user experience.

# **Algorithm and Logic**

The Weather App is designed to provide users with real-time weather data based on their input (city name) by interacting with the OpenWeatherMap API. The overall flow of the application is divided into a series of steps, from user input to API interaction and displaying results. The logic is structured to ensure smooth user interaction, handle errors gracefully, and present the weather data in an easy-to-understand format.

Below is a detailed explanation of the algorithm and logic that powers the Weather App, followed by a **pseudocode** and **flowchart** which provide a high-level overview of the program’s operation.

**1. Initialization**

* The application initializes the user interface (UI) elements, such as the city input field, labels for displaying the weather data, and the button to trigger the weather fetch action.
* The WeatherApp class contains an \_\_init\_\_() method, which sets up all the UI components. This method also connects the **button click event** to the get\_weather() method, allowing users to fetch the weather when they press the button.

**2. User Input**

* The user is prompted to enter the name of a city in the QLineEdit widget.
* Upon clicking the “Get Weather” button, the program takes the text entered in the input field and processes it using the city\_input.text() method. The input is then **trimmed** to remove leading and trailing spaces using the .strip() function.

**3. Input Validation**

* If the user does not enter a city name and the input field is empty, the app will display an error message, alerting the user that the city name cannot be empty. This is done using the display\_error() method.

**4. Fetching Data from API**

* If a valid city name is entered, the app sends a request to the OpenWeatherMap API using the requests.get() method.
  + The URL of the request includes the city name, the app’s API key, and the unit of measurement for temperature (imperial, in this case).
  + The app waits for the response from the API and handles the HTTP request asynchronously.
  + The get\_weather() method processes the response. If the request is successful, the app parses the JSON data to extract key weather information such as temperature, weather description, and weather ID (used for emoji representation).

**5. Displaying Weather Data**

* Once the data is fetched and parsed, the app updates the UI to display the temperature (in Fahrenheit), the weather description (e.g., “Clear sky”), and an emoji representing the weather condition (such as a sun for clear skies, cloud for overcast, or snowflake for snow).
  + The temperature is displayed with the temperature\_label.
  + The weather condition is displayed with the description\_label.
  + The emoji is displayed with the emoji\_label.
  + The display is updated using the display\_weather() method.

**6. Handling Errors**

* If the API request fails or the response does not contain the expected data, the app will display an error message.
  + Errors such as network issues or invalid city names are handled by the display\_error() method, which updates the UI to notify the user of the issue.

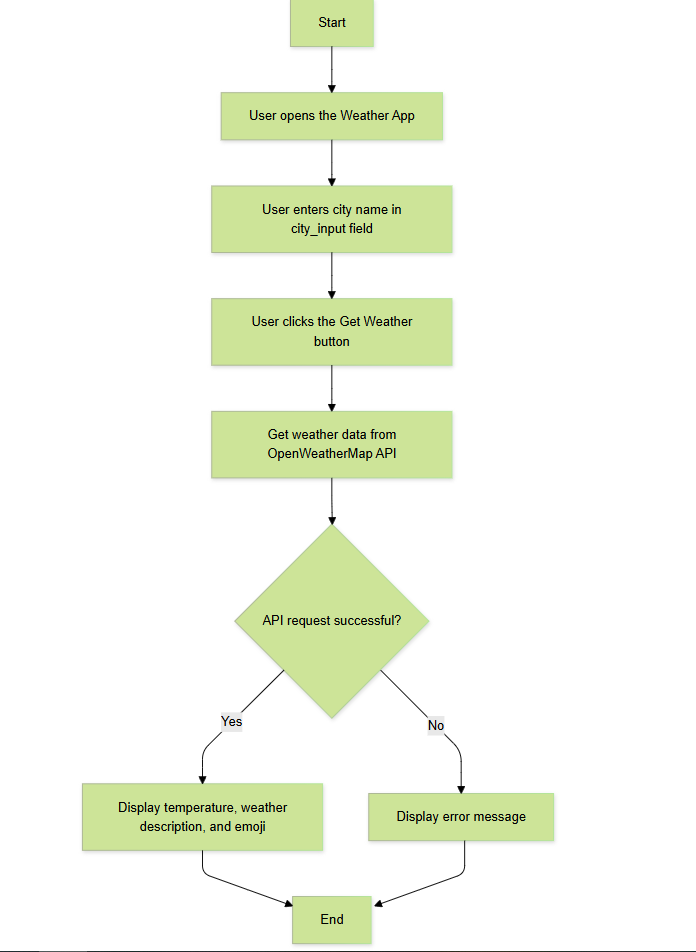
**7. Weather Emoji Mapping**

* A crucial part of the application is the **weather emoji** representation. The weather emoji is determined based on the weather ID provided by the API. The get\_weather\_emoji() method maps different weather IDs to their corresponding emoji, which helps to visualize the weather condition.

**8. Termination**

* After the weather data is displayed or an error is shown, the program continues to wait for new user input. The event loop will keep running, waiting for the user to interact with the app again (e.g., by entering a new city name).

## **FLOWCHART**



# **OOP in Action: Detailed Class Explanation**

The Weather App is built using **Object-Oriented Programming (OOP)** principles, which promote code reusability, modularity, and maintainability. Below is a detailed explanation of how OOP concepts are applied within the main class, WeatherApp.

**Class: WeatherApp**

The WeatherApp class is the central part of the application. It is responsible for setting up the user interface (UI), handling user inputs, making API requests to fetch weather data, and displaying the results. The class also manages error handling and other necessary operations.

**Constructor (\_\_init\_\_)**

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.city\_label = QLabel("Enter City Name", self)

self.city\_input = QLineEdit(self)

self.get\_weather\_button = QPushButton("Get Weather", self)

self.temperature\_label = QLabel(self)

self.emoji\_label = QLabel(self)

self.description\_label = QLabel(self)

self.initUI()

* **Purpose**: The constructor initializes the WeatherApp class, sets up the UI components (labels, input field, button), and calls the initUI() method to configure the layout.
* **super().\_\_init\_\_()**: This calls the constructor of the parent class (QWidget) to initialize the base window properties, like positioning and sizing.
* **UI Components**: The constructor creates multiple UI elements:
  + **QLabel**: Displays static text or dynamic content, like weather data and error messages.
  + **QLineEdit**: A text field for user input, specifically the city name.
  + **QPushButton**: A button to trigger the weather-fetching action when clicked.
* **initUI() Method**: The constructor also invokes the initUI() method to configure the layout and appearance of the UI components.

**initUI() Method**

def initUI(self):

self.setWindowTitle("Weather App")

vbox = QVBoxLayout()

vbox.addWidget(self.city\_label)

vbox.addWidget(self.city\_input)

vbox.addWidget(self.get\_weather\_button)

vbox.addWidget(self.temperature\_label)

vbox.addWidget(self.emoji\_label)

vbox.addWidget(self.description\_label)

self.setLayout(vbox)

* **Purpose**: This method sets up the layout and appearance of the application's UI. It organizes the widgets vertically using QVBoxLayout and adds each widget to the layout in the desired order.
* **Setting Window Title**: self.setWindowTitle("Weather App") sets the title of the application window to "Weather App."
* **Widgets Arrangement**: The QVBoxLayout ensures that the widgets are placed vertically, making it easier for the user to interact with them sequentially.

**get\_weather() Method**

def get\_weather(self):

api\_key = "078c127dd1b35bbb740a4908320c5fa3"

city = self.city\_input.text().strip()

if not city:

self.display\_error("City name cannot be empty")

return

url="http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api\_key}&units=imperi"

try:

response = requests.get(url)

response.raise\_for\_status()

data = response.json()

self.display\_weather(data)

except requests.exceptions.RequestException as e:

self.display\_error(f"Error: {e}")

except KeyError:

self.display\_error("Unexpected data format from API")

* **Purpose**: This method handles the core functionality of fetching weather data. When the user clicks the "Get Weather" button, this method is triggered.
* **City Name**: The city\_input.text().strip() retrieves the city name entered by the user and removes any leading or trailing spaces.
* **API Request**: A request is made to the OpenWeatherMap API using the entered city name. The response is checked, and if successful, the weather data is extracted in JSON format.
* **Error Handling**: If the request fails or if the data format is unexpected, the app handles the exceptions by calling self.display\_error().

**display\_weather() Method**

python

CopyEdit

def display\_weather(self, data):

temperature\_f = data["main"]["temp"]

weather\_id = data["weather"][0]["id"]

weather\_description = data["weather"][0]["description"]

self.temperature\_label.setText(f"{temperature\_f:.2f}°F")

self.emoji\_label.setText(self.get\_weather\_emoji(weather\_id))

self.description\_label.setText(weather\_description.capitalize())

* **Purpose**: This method is responsible for updating the UI with the fetched weather data.
* **Extracting Weather Data**: The temperature, weather ID, and description are extracted from the JSON response returned by the API. This data is used to update the UI.
* **Displaying Data**:
  + The temperature is displayed in Fahrenheit with two decimal points using the setText() method.
  + An appropriate weather emoji is fetched using the get\_weather\_emoji() method, based on the weather ID.
  + The description is capitalized and displayed under the temperature.

**display\_error() Method**

def display\_error(self, message):

self.temperature\_label.setStyleSheet("font-size: 30px; color: red;")

self.temperature\_label.setText(message)

self.emoji\_label.setText("")

self.description\_label.setText("")

* **Purpose**: This method is used to display error messages on the UI if something goes wrong.
* **Updating UI**: If there’s an error (such as an empty city name or failed API request), this method:
  + Changes the text color to red.
  + Displays the error message in the temperature label.
  + Clears out the emoji and description labels to avoid displaying old or incorrect data.

**get\_weather\_emoji() Method**

python

CopyEdit

@staticmethod

def get\_weather\_emoji(weather\_id):

match weather\_id:

case 200 | 201 | 202 | 210 | 211 | 212 | 221 | 230 | 231 | 232:

return "⛈️"

case 300 | 301 | 302 | 310 | 311 | 312 | 313 | 314 | 321:

return "🌧️"

case 500 | 501 | 502 | 503 | 504 | 511 | 520 | 521 | 522 | 531:

return "🌧️"

case 600 | 601 | 602 | 611 | 612 | 613 | 615 | 616 | 620 | 621 | 622:

return "❄️"

case 701 | 711 | 721 | 731 | 741 | 751 | 761 | 762 | 771 | 781:

return "🌫️"

case 800:

return "☀️"

case 801:

return "🌤️"

case 802:

return "⛅"

case 803 | 804:

return "☁️"

case \_:

return "❓"

* **Purpose**: This method determines the appropriate emoji to display based on the weather condition (represented by a weather ID from the OpenWeatherMap API).
* **Matching IDs**: The match statement is used to check the weather ID and return the corresponding emoji. For example:
  + If the weather ID corresponds to rain (IDs 500-531), it returns the rain emoji 🌧️.
  + If the weather ID corresponds to clear skies (ID 800), it returns the sun emoji ☀️.

# **Future Implementation:**

**Dark Mode**: Add support for dark mode to make the app more user-friendly during the night.

**Better UI Design**: Improve the user interface with a more modern design, animations, or interactive elements.

**Notifications**: Add push notifications or alerts for severe weather conditions based on the forecast.

**Geolocation**: Instead of requiring the user to input a city, implement automatic geolocation detection to fetch weather based on the user’s current location.

# **CONCLUSION:**

In this project, we have successfully created a weather app using Python and PyQt5 that retrieves real-time weather information from the OpenWeatherMap API. By leveraging Object-Oriented Programming (OOP) principles, we’ve built a modular and maintainable application with clean and structured code. The app is fully functional with basic error handling and UI elements, providing an intuitive interface for users to check the weather. With future improvements and extensions, the app can evolve into a more feature-rich and user-friendly tool.