Journalify

A desktop based

Daily Journaling Application

Course No: CSE 3106

Course Title: Software Development Project

Submitted to:	Submitted by:
Dr. Amit Kumar Mondal Associate Professor Computer Science and Engineering Discipline, Khulna University,Khulna	Name: Uma Datta Student ID: **210226** & Name:Shaikh Jaberul Islam Moin Student ID: **210238**

Code Review of Project Title: Weather Application

Project Holders:

Name: Abrar Jahin
Student ID:210236
Name: Hasibul Hasan
Student ID:210241

Code Reviews

1.Code smells:

(a)Large or complex Method:

```
API Request Handling:
response = requests.get(api_url)
dict_content = response.content.decod


JSON Parsing:
temperature = dict_content["main"]["temp"] - 273.16
Condition = dict_content["weather"][0]["description"].capitalize()
Humidity = dict_content["main"]["humidity"]
pressure = dict_content["main"]["pressure"]
wind = dict_content["wind"]["speed"]


Error Handling:
except:
 print("Sorry, You Entered Wrong Location.")getLoc = loc.geocode(locatio


Exception Handling:
try ...
except:
 print("Sorry, You Entered Wrong Location.")
.....
def weather_data(location=None):
    loc = Nominatim(user_agent="GetLoc")
    # print("Enter Your Location : ", end=" ")
    # location = input()
```

```
    getLoc = loc.geocode(location)
    try:
        Latitude = getLoc.latitude
        Longitude = getLoc.longitude

        # print(f"Latitude : {Latitude},Longitude : {Longitude}")
        api_url =
f"https://api.openweathermap.org/data/2.5/weather?lat={Latitude}&lon={Lo
ngitude}&appid=313c138faa75abd252dd9aefcd356d9c"
        response = requests.get(api_url)
        dict_content = response.content.decode("utf-8")
        # print(dict_content)
        temperature = dict_content["main"]["temp"] - 273.16

        # print(f'Country         : {dict_content["sys"]["country"]
        # print(f"Place           : {location.capitalize()}")
        Temperature = round(temperature, 2)
        Condition =
dict_content["weather"][0]["description"].capitalize()
        Humidity = dict_content["main"]["humidity"]
        pressure = dict_content["main"]["pressure"]
        wind = dict_content["wind"]["speed"]
        return Temperature, Condition, Humidity, pressure, wind
    except:
        print("Sorry,You Entered Wrong Location.")
```

More concise and simplified code:

Reasons:

- Reduced Complexity
- Fewer Lines
- Improved Readability
- Direct Return of Weather Data.

(b) Excessive Comment:

```
# Weather Application

from geopy.geocoders import Nominatim
import requests
import json


def weather_data(location=None):
    loc = Nominatim(user_agent="GetLoc")
    # print("Enter Your Location : ", end=" ")
    # location = input()

    getLoc = loc.geocode(location)
    try:
        Latitude = getLoc.latitude
        Longitude = getLoc.longitude

        # print(f"Latitude : {Latitude},Longitude : {Longitude}")
        api_url =
f"https://api.openweathermap.org/data/2.5/weather?lat={Latitude}&lon={Lo
ngitude}&appid=313c138faa75abd252dd9aefcd356d9c"
        response = requests.get(api_url)
        dict_content = response.content.decode("utf-8")
        # print(dict_content)
        temperature = dict_content["main"]["temp"] - 273.16

        # print(f'Country         : {dict_content["sys"]["country"]
        # print(f"Place           : {location.capitalize()}")
        Temperature = round(temperature, 2)
        Condition =
dict_content["weather"][0]["description"].capitalize()
        Humidity = dict_content["main"]["humidity"]
        pressure = dict_content["main"]["pressure"]
        wind = dict_content["wind"]["speed"]
        return Temperature, Condition, Humidity, pressure, wind
    except:
```

```
        print("Sorry,You Entered Wrong Location.")

# weather_data()
```

The code without the comments:

Reasons:

- Readable Variable and Functions name
- Clear flow from top to bottom
- Use of built in functions that don't require any additional explanation.

(c) Too many if/else statement:

```
for l in lines:
    m2, u2, pass2 = l.split(",")
    if u == u2:
        password = password + "\n"
        if password != pass2:
            tkinter.messagebox.showwarning(
                title=Warning, message="Incorrect Password"
            )
            flag = 2
            break
        if password == pass2:
            flag = 1
            break
if flag == 0:
    tkinter.messagebox.showwarning(title=Warning, message="User
Does not exist")
file.close()
if flag == 1:
```

```
    root.destroy()
    with open("Client.py") as app:
        exec(app.read())
```

Improves by simplifying:

Reasons:

- The code becomes more readable
- Multiple if-else statement for different scenarios leading into redundancy,so the scenarios are consolidated into a single dictionary,eliminating the retitive code
- Using a dictionary allows easy modification and additions without cluttering the main code.

(d) Duplicate Code:

```
label1 = Label(
    root, text="WIND", font=("Helvetica", 15, "bold"), fg="white",
bg="#1ab5ef"
)
label1.pack(padx=5, pady=5, side=BOTTOM)
label1.place(x=150, y=400)

label2 = Label(
    root, text="HUMIDITY", font=("Helvetica", 15, "bold"),
fg="white", bg="#1ab5ef"
)
label2.pack(padx=5, pady=5, side=BOTTOM)
label2.place(x=270, y=400)

label3 = Label(
    root,
    text="DESCRIPTION",
    font=("Helvetica", 15, "bold"),
```

```
    fg="white",
    bg="#1ab5ef",
)
label3.pack(padx=5, pady=5, side=BOTTOM)
label3.place(x=440, y=400)

label4 = Label(
    root, text="PRESSURE", font=("Helvetica", 15, "bold"),
fg="white", bg="#1ab5ef"
)
label4.pack(padx=5, pady=5, side=BOTTOM)
label4.place(x=650, y=400)
```

```
# List of label attributes
label_attributes = [
    ("WIND", 150),
    ("HUMIDITY", 270),
    ("DESCRIPTION", 440),
    ("PRESSURE", 650)
]

# Create labels dynamically
weather_labels = []
for text, x_position in label_attributes:
    label = Label(
        root, text=text, font=("Helvetica", 15, "bold"),
fg="white", bg="#1ab5ef"
    )
    label.pack(padx=5, pady=5, side=BOTTOM)
    label.place(x=x_position, y=400)
    weather_labels.append(label)
```

The simplified version of the create_widgets method:

Reasons:

- Modularization
- Each method is responsible for a specific aspect of the user interface,promoting separation of principle
- Elimination of Global Variables
- The class encapsulates the entire functionality of the weather application, making it easier to reuse the code in other projects or to integrate with other parts of the application.

(e) Magic numbers or hard-coded values:

```
"E://Study//3-1//Software Development Project//Weather
Application//DEV//Software-Development-Project//Model//Accounts.tx
t"
api_url =
f"https://api.openweathermap.org/data/2.5/weather?lat={Latitude}&l
on={Longitude}&appid=313c138faa75abd252dd9aefcd356d9c"
.....


user.insert(0, "Username")
code.insert(0, "Password")
user.place(x=30, y=80)
code.place(x=30, y=150)
root.geometry("925x500+300+200")
root.geometry("900x500+300+200")
textfield.place(x=38, y=47)
myimage_icon.place(x=392, y=39)
logo.place(x=150, y=100)
```

```
label1.place(x=150, y=400)
label2.place(x=270, y=400)
label3.place(x=440, y=400)
label4.place(x=650, y=400)
t.place(x=400, y=150)
c.place(x=400, y=250)
w.place(x=160, y=430)
h.place(x=300, y=430)
d.place(x=490, y=430)
p.place(x=700, y=430)
root.geometry("925x500+300+200")
Label(root, image=img, bg="white").place(x=50, y=50)
frame = Frame(root, width=350, height=350, bg="white")
frame.place(x=480, y=70)
heading.place(x=100, y=5)
user.place(x=30, y=80)
Frame(frame, width=295, height=2, bg="black").place(x=25, y=107)
code.place(x=30, y=150)
Frame(frame, width=295, height=2, bg="black").place(x=25, y=177)
Button(
    frame,
    width=39,
    pady=7,
    text="Sign in",
    bg="#57a1f8",
    fg="white",
    border=0,
    command=button_clicked_in,
).place(x=35, y=204)
label.place(x=75, y=270)
sign_up.place(x=215, y=270)
root.geometry("925x500+300+200")
Label(root, image=img, bg="white").place(x=50, y=50)
frame = Frame(root, width=350, height=350, bg="white")
frame.place(x=480, y=70)
heading.place(x=100, y=5)
mail.place(x=30, y=80)
Frame(frame, width=295, height=2, bg="black").place(x=25, y=100)
user.place(x=30, y=120)
```

```
Frame(frame, width=295, height=2, bg="black").place(x=25, y=140)
code.place(x=30, y=160)
Frame(frame, width=295, height=2, bg="black").place(x=25, y=180)
Button(
    frame,
    width=39,
    pady=7,
    text="Sign Up",
    bg="#57a1f8",
    fg="white",
    border=0,
    command=button_clicked,
).place(x=35, y=200)
```

The simplified version:

Reasons:

- Define Constants: Define constants for file paths, default text, coordinates, dimensions, and any other hard-coded values.
- Replace Hard-coded Values: Replace the hard-coded values with the defined constants throughout the code.

(f) Error Handling:

```
try:
   # Code that may raise exceptions
   response = requests.get(api_url)
   dict_content = response.json()
except requests.exceptions.RequestException as e:
   # Handle network-related errors
   print("Error making request:", e)
except json.JSONDecodeError as e:
   # Handle JSON decoding errors
   print("Error decoding JSON response:", e)
except Exception as e:
   # Handle other unexpected errors
   print("An unexpected error occurred:", e)
```

```
try:
   # Code that may raise exceptions
   result = obj.timezone_at(lng=location.longitude, lat=location.latitude)
except Exception as e:
   # Handle the error and display a custom error message
   print("Error:", e)
   messagebox.showerror("Error", "An error occurred while retrieving
timezone information.")
```

```
file = None
try:
   file = open("example.txt", "r")
   # Code that reads from the file
except FileNotFoundError as e:
   print("File not found:", e)
finally:
   if file:
      file.close()
```

```
import logging

logging.basicConfig(filename='example.log', level=logging.ERROR)

try:
   # Code that may raise exceptions
except Exception as e:
   logging.error("An error occurred: %s", e)
```

2. Code formatting:

- **Indentation:** All lines within each block of code are uniformly indented.
- **Spacing:** Proper spacing is maintained around operators, commas, and other elements to improve readability. For example, there are spaces around assignment operators (=), commas (,), and arithmetic operators (+, -). Additionally, consistent spacing is maintained between different elements to enhance clarity.
- **Consistent Line Length:** Lines are wrapped appropriately to fit within a reasonable line length, typically recommended to be around 80-100 characters. This avoids horizontal scrolling and makes the code easier to read without the need to scroll sideways.
- **Comments:** There are some comments in the codes which are not actually needed.

- **Overall Structure:** The overall structure of the code, including the arrangement of class definitions, method definitions, and the main execution block, is organized and presented in a logical and readable manner.

3.Architecture:

➔ The defined Architecture: MVC pattern

Yes, the MVC (Model-View-Controller) architecture is a suitable choice for a weather application, depending on its complexity and requirements. Though they chose client server architecture in the initial design.

Model: The Model represents the data and business logic of the application. In this weather application, the Model would handle tasks such as fetching weather data from external APIs, processing the data, and storing it if necessary. This component encapsulates all the operations related to weather data manipulation.

View: The View is responsible for presenting the data to the user and handling user interactions. In this weather application, the View would include the user interface elements such as the display of weather information, input fields for location or settings, and any visual elements like charts or maps representing weather patterns.

Controller: The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes it, and updates the Model accordingly. In this weather application, the Controller would handle user actions such as requesting weather data for a specific location, updating user preferences, or triggering data refreshes.

➔ Separation of Concerns:

Using MVC separates the application into distinct components. It splits into its respective components.

➔ Code is in sync with code pattern:

The provided code explicitly follows the MVC (Model-View-Controller) architecture. However, we can map some elements of the code to MVC components:

View: The user interface elements such as Entry, Button, Label, and their placements define the View component. They are responsible for displaying data to the user and handling user interactions.
Controller: In the provided code, the get_weather() method is supposed to be considered as part of the Controller. It responds to user actions (such as clicking the search button) and triggers the logic to fetch weather data.
Model: The Model represents the data and business logic. However, in this complete MVC architecture, this part would handle tasks like fetching weather data from an external API, parsing the response, and updating the View accordingly.

4.Non Functional requirements:

➔ **Code Readability:**
- The code demonstrates decent readability with descriptive variable names and straightforward logic.

➔ **Consistency:**
- The code exhibits sometimes consistent naming conventions and formatting, contributing to its maintainability.
- Maintaining consistency in coding style, variable naming, and architectural patterns throughout the codebase helps us to navigate and modify the code more efficiently.

➔ **Error Handling:**

- Proper error handling mechanisms are missing in the provided code. Incorporating error handling to gracefully handle unexpected scenarios, such as network failures or invalid user input, is crucial for maintainability.
- Error messages and logging can provide valuable insights during debugging and maintenance activities.

➔ **Testing:**

- The code does not include automated tests to verify its functionality. Implementing unit tests, integration tests, and possibly end-to-end tests can help ensure the correctness of the code and facilitate future changes with confidence.
- Test-driven development (TDD) practices can guide the development process and improve the maintainability of the codebase by promoting modular design and testability.

➔ **Performance:**

- Responsiveness: The code provides a user interface with an entry field to input a city and a button to trigger weather data retrieval. It should respond quickly to user interactions.
- Efficiency: The efficiency of data retrieval and processing is not directly addressed in the provided code. It depends on the implementation of the get_weather() method, including the efficiency of fetching weather data from external sources.

➔ **Usability:**

- User Interface Design: The user interface is simple and intuitive, consisting of an entry field and a button for weather data input, along with labels for displaying weather information.
- Accessibility: There are no explicit accessibility features implemented in the provided code. Accessibility considerations should be taken into account to ensure the application is usable by all users, including those with disabilities.

➔ **Security:**

- Data Protection: The code does not involve handling sensitive user data. However, if the application were to evolve to handle user authentication or personalization, appropriate security measures should be implemented to protect user data.
- Input Validation: The code should validate user input (e.g., city names) to prevent injection attacks or invalid data causing unexpected behavior.

➔ **Scalability:**

- Concurrency: The code does not address concurrency or parallel processing explicitly. Depending on the implementation of the

get_weather() method, it may support concurrent requests or require modifications for scalability.

Let's assess the provided code for modularity:

- **Component Separation:**
  - The code exhibits some degree of component separation. It divides the user interface elements (such as Entry, Button, and Label) into separate entities.
  - However, there is no clear separation of concerns between different functional aspects of the application, such as data retrieval, processing, and presentation. These concerns are somewhat intertwined within the same class.
- **Encapsulation:**
  - The code encapsulates the graphical user interface (GUI) elements within the WeatherApp class. This encapsulation makes it easier to manage the GUI components as a single unit.
  - However, there is limited encapsulation of other functionalities, such as weather data retrieval and processing. These functionalities could benefit from encapsulation within separate classes or modules.
- **Reusability:**
  - The code does not demonstrate significant reusability. Each component, such as the search button or weather labels, is specific to the weather application and not designed for general-purpose reuse.
  - To improve reusability, it could refactor common functionalities, such as GUI components or data retrieval utilities, into separate modules that could be reused across different applications.

- **Dependencies:**
  - There is no unnecessary dependencies in the application
- **Scalability:**
  - Breaking down the application into smaller, modular components with well-defined interfaces can improve scalability by allowing for easier maintenance, extension, and parallel development.

So,This is the code review of the Weather App.