EXPERIMENT – 3
CLI, GUI AND VUI USING PYTHON

AIM:

To design and implement a Python application using three different user interfaces:Command Line Interface (CLI), Graphical User Interface (GUI), and Voice User Interface (VUI) and to understand how users interact with software through text-based, graphical, and voice-based methods.

TOOLS REQUIRED:

• Python 3.x
• VS Code
• Windows Operating System
• Command Prompt / VS Code Terminal

1. CLI IMPLEMENTATION (Command Line Interface):

THEORY:

A Command Line Interface (CLI) allows users to interact with a program by typing text-based commands in the terminal. CLI applications are lightweight, fast, and suitable for system-level operations.

In this experiment, an Expense Tracker application is developed using Python. The program uses file handling to permanently store expense records in a text file. Each expense entry includes an ID, amount, category, and date.

CODE:

```
import sys
import os
from datetime import datetime
FILENAME = "expenses.txt"
if not os.path.exists(FILENAME):
```

```
    open(FILENAME, "w").close()
def get_next_id():
    with open(FILENAME, "r") as file:
        lines = file.readlines()
        if not lines:
            return 1
        last_line = lines[-1]
        return int(last_line.split(",")[0]) + 1
def add_expense(amount, category):
    expense_id = get_next_id()
    date = datetime.now().strftime("%Y-%m-%d")
    with open(FILENAME, "a") as file:
        file.write(f"{expense_id},{amount},{category},{date}\n")
    print("Expense added successfully!")
def list_expenses():
    with open(FILENAME, "r") as file:
        lines = file.readlines()
    if not lines:
        print("No expenses found.")
        return
    print("ID | Amount | Category | Date")
    print("-" * 35)
    for line in lines:
        expense_id, amount, category, date = line.strip().split(",")
        print(f"{expense_id} | {amount} | {category} | {date}")

def total_expenses():
    total = 0
    with open(FILENAME, "r") as file:
        for line in file:
            amount = float(line.strip().split(",")[1])
            total += amount
    print(f"Total Expenses: {total}")
def delete_expense(expense_id):
    with open(FILENAME, "r") as file:
        lines = file.readlines()
    with open(FILENAME, "w") as file:
        for line in lines:
            if not line.startswith(str(expense_id) + ","):
                file.write(line)
```

```
    print("Expense deleted successfully!")
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage:")
        print("  python expense.py add <amount> <category>")
        print("  python expense.py list")
        print("  python expense.py total")
        print("  python expense.py delete <id>")
        sys.exit()
    command = sys.argv[1]
    if command == "add" and len(sys.argv) == 4:
        add_expense(sys.argv[2], sys.argv[3])
    elif command == "list":
        list_expenses()
    elif command == "total":
        total_expenses()
    elif command == "delete" and len(sys.argv) == 3:
        delete_expense(sys.argv[2])
    else:
        print("Invalid command or arguments.")
```

HOW IT WORKS :

1. The program is a CLI-based Expense Tracker application.
2. The expenses are stored in a file named `expenses.txt`.
3. Each expense entry contains:
   - ID
   - Amount
   - Category
   - Date
4. The `add_expense()` function: Adds a new expense to the file with an auto-generated ID and current date.
5. The `list_expenses()` function:Displays all stored expenses in a formatted table.
6. The `total_expenses()` function:Calculates and displays the total amount spent.
7. The `delete_expense()` function:Removes an expense based on the given ID.
8. File handling ensures that all expense data is stored permanently.

9. The `sys.argv` method allows command-line arguments to control program functionality.
10. Exception handling and conditional checks prevent invalid command usage.

OUTPUT:

```
OUTPUT    PROBLEMS    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py add 300 Food
Expense added successfully!
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py add 300 Food
Expense added successfully!
Expense added successfully!
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py add 200 Food
Expense added successfully!
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py list
ID | Amount | Category | Date
---------------------------------
1 | 300 | Food | 2026-02-17
2 | 200 | Food | 2026-02-17
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py total
Total Expenses: 500.0
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha> python cli.py delete 1
Expense deleted successfully!
PS C:\Users\SHRUTI K\OneDrive\Desktop\UserInterface_subha>
```

2. GRAPHICAL USER INTERFACE(GUI):

THEORY:

A Graphical User Interface (GUI) allows users to interact with a program using windows, buttons, and graphical elements instead of typing commands.

In this experiment, a Pomodoro Timer application is developed using Tkinter. The application follows the Pomodoro technique, where the user works for 25 minutes and takes a 5-minute break. The timer automatically switches between work and break sessions.Event-driven programming is used to handle button clicks and timer updates.

CODE:

```
import tkinter as tk
WORK_TIME = 25 * 60
BREAK_TIME = 5 * 60
```

```
time_left = WORK_TIME
is_running = False
is_work_time = True
sessions_completed = 0
def update_timer():
    global time_left, is_running, is_work_time, sessions_completed
    if is_running and time_left > 0:
        minutes = time_left // 60
        seconds = time_left % 60
        timer_label.config(text=f"{minutes:02}:{seconds:02}")
        time_left -= 1
        root.after(1000, update_timer)
    elif is_running and time_left == 0:
        if is_work_time:
            sessions_completed += 1
            session_label.config(text=f"Sessions Completed:
{sessions_completed}")
            switch_to_break()
        else:
            switch_to_work()
def start_timer():
    global is_running
    if not is_running:
        is_running = True
        update_timer()
def pause_timer():
    global is_running
    is_running = False
def reset_timer():
    global time_left, is_running, is_work_time
    is_running = False
    is_work_time = True
    time_left = WORK_TIME
    timer_label.config(text="25:00")
    status_label.config(text="Work Time", fg="red")
    root.config(bg="white")
def switch_to_break():
    global time_left, is_work_time
    is_work_time = False
    time_left = BREAK_TIME
```

```
    status_label.config(text="Break Time", fg="green")
    root.config(bg="#d4edda")
    update_timer()
def switch_to_work():
    global time_left, is_work_time
    is_work_time = True
    time_left = WORK_TIME
    status_label.config(text="Work Time", fg="red")
    root.config(bg="white")
    update_timer()
root = tk.Tk()
root.title("Pomodoro Timer")
root.geometry("400x300")
root.resizable(False, False)

status_label = tk.Label(root, text="Work Time", font=("Arial", 18),
fg="red")
status_label.pack(pady=10)

timer_label = tk.Label(root, text="25:00", font=("Arial", 48, "bold"))
timer_label.pack(pady=10)

session_label = tk.Label(root, text="Sessions Completed: 0",
font=("Arial", 12))
session_label.pack(pady=5)

button_frame = tk.Frame(root)
button_frame.pack(pady=20)

start_button = tk.Button(button_frame, text="Start", width=10,
command=start_timer)
start_button.grid(row=0, column=0, padx=5)
pause_button = tk.Button(button_frame, text="Pause", width=10,
command=pause_timer)
pause_button.grid(row=0, column=1, padx=5)
reset_button = tk.Button(button_frame, text="Reset", width=10,
command=reset_timer)
reset_button.grid(row=0, column=2, padx=5)
root.mainloop()
```

HOW IT WORKS :

The program creates a window using Tkinter.

• A timer label displays the countdown time.

• The Start button begins the countdown.

• The Pause button stops the timer temporarily.

• The Reset button resets the timer to 25 minutes.

• When the timer reaches 0, it automatically switches between Work Time and Break Time.

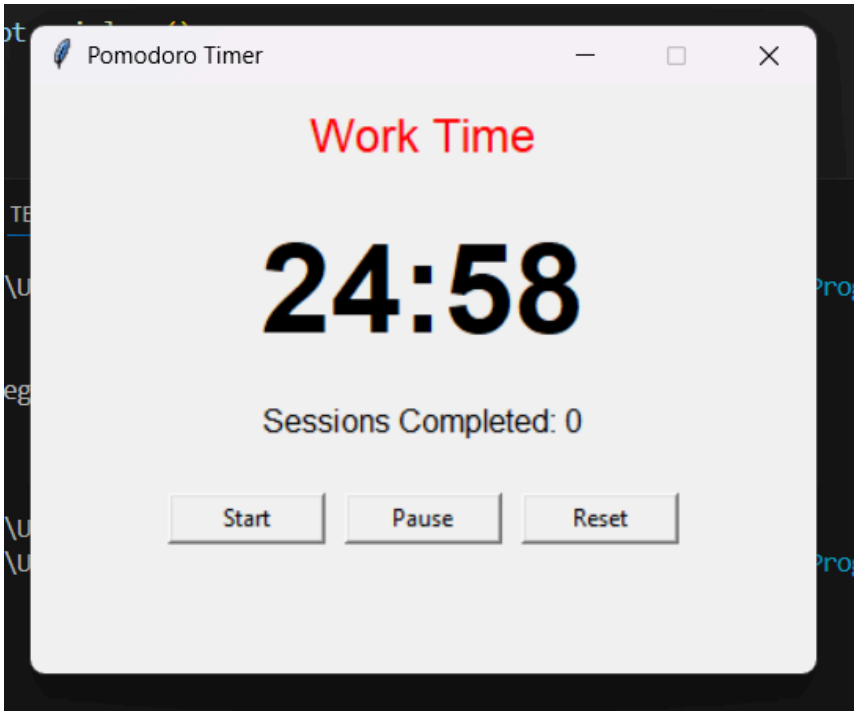
• A session counter keeps track of completed work sessions.

• The background color changes based on work or break mode.

• The `after()` method is used to update the timer every second.

• The program runs until the user closes the window

Output:



3.VOICE USER INTERFACE(VUI):

THEORY:

A Voice User Interface (VUI) allows users to interact with a system using voice commands instead of typing or clicking.

CODE:

```
import speech_recognition as sr
import pyttsx3
import datetime
import webbrowser
import os

engine = pyttsx3.init()
def speak(text):
    engine.say(text)
    engine.runAndWait()

def listen():
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("Listening...")
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

    try:
        command = recognizer.recognize_google(audio)
        print("You said:", command)
        return command.lower()
    except sr.UnknownValueError:
        speak("Sorry, I did not understand.")
        return ""
    except sr.RequestError:
        speak("Network error.")
        return ""
def run_assistant():
    speak("Hello! I am your assistant.")

    while True:
```

```
        command = listen()
        if "time" in command:
            time = datetime.datetime.now().strftime("%I:%M %p")
            speak("The time is " + time)
        elif "open notepad" in command:
            speak("Opening Notepad")
            os.system("notepad")
        elif "open youtube" in command:
            speak("Opening YouTube")
            webbrowser.open("https://www.youtube.com")
        elif "exit" in command or "stop" in command:
            speak("Goodbye!")
            break
        elif command == "":
            continue
        else:
            speak("Command not recognized.")
if __name__ == "__main__":
    run_assistant()
```

HOW IT WORKS:

• The program uses the microphone to listen to the user's voice command.

• The `SpeechRecognition` library converts the spoken voice into text.

• The recognized text is processed and checked for specific keywords.

• Based on the command, the program performs actions such as:

- Telling the current time
- Opening applications
- Opening websites

• The `pyttsx3` library is used to convert text responses into speech.

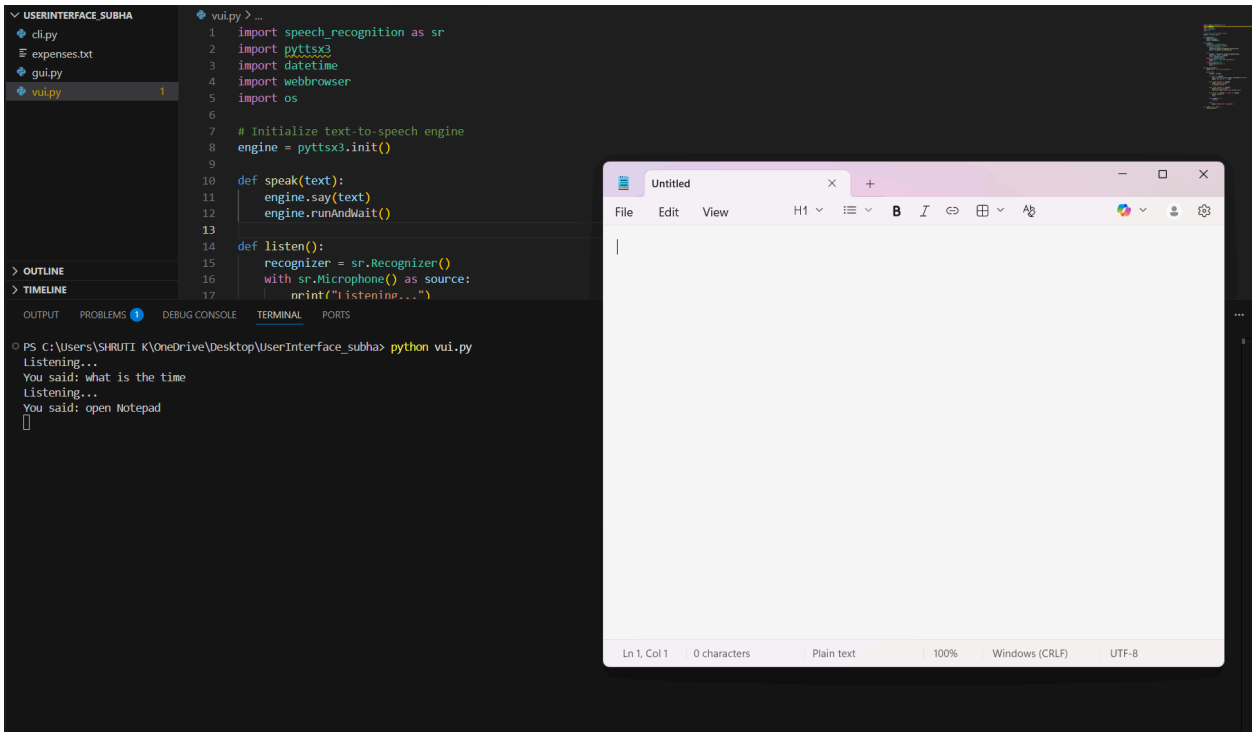
• The assistant speaks back the response to the user.

• A loop allows the assistant to continuously listen until the user says "exit" or "stop".

• Exception handling manages errors like unclear speech or network issues.

• The program stops when the exit command is given.

OUTPUT:



RESULT:

The CLI, GUI, and VUI versions of the program were successfully implemented and tested. All interfaces performed the required operations correctly. The CLI allowed text-based interaction, the GUI provided a user-friendly graphical interface, and the VUI enabled voice-based interaction. Each interface demonstrated proper functionality and error handling.