Gabriel Q. Escobido

BSCoE – 2A

SOFTWARE DESIGN

Laboratory Exercise No. 3 CH2

Title: Exploring Programming Paradigms

Brief Introduction

Programming paradigms define the style and structure of writing software programs. This exercise introduces imperative, object-oriented, functional, declarative, event-driven, and concurrent programming paradigms and their applications.

Objectives

- Compare and contrast different programming paradigms.
- Implement examples using Python for multiple paradigms.
- Explore practical use cases for each paradigm.

Detailed Discussion

Programming Paradigm	Description	Example Applications
Imperative	Focuses on step-by-step instructions.	Low-level programming tasks
Object-Oriented	Organizes code using objects and classes.	GUI applications, simulations
Functional	Emphasizes mathematical functions and immutability.	Data analysis, AI
Declarative	Specifies what to do without describing how to do it.	SQL, configuration files
Event-Driven	Responds to events like clicks, signals, or messages.	GUIs, games
Concurrent	Manages multiple computations at the same time.	Web servers, parallel processing

Materials

- Python environment
- VS Code IDE

Procedure

1. Implement imperative programming in Python:

```
# Imperative programming example
nums = [1, 2, 3, 4, 5]
total = 0
for num in nums:
  total += num
print("Total:", total)
```

1. Create a simple object-oriented program:

```
class Person:
  def __init__(self, name, age):
      self.name = name
      self.age = age

  def greet(self):
      return f"Hello, my name is {self.name} and I am {self.age} years old."

p = Person("Alice", 25)
print(p.greet())
```

1. Write a functional programming example using Python's map and filter:

```
nums = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, nums))
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print("Squared Numbers:", squared)
print("Even Numbers:", even_nums)
```

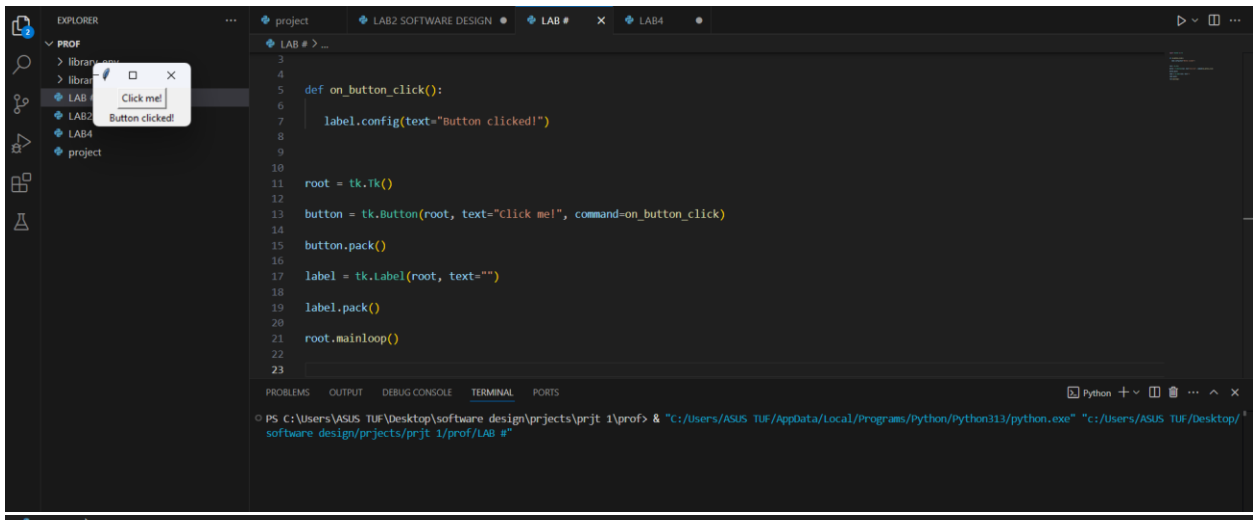
1. Showcase event-driven programming using Tkinter:

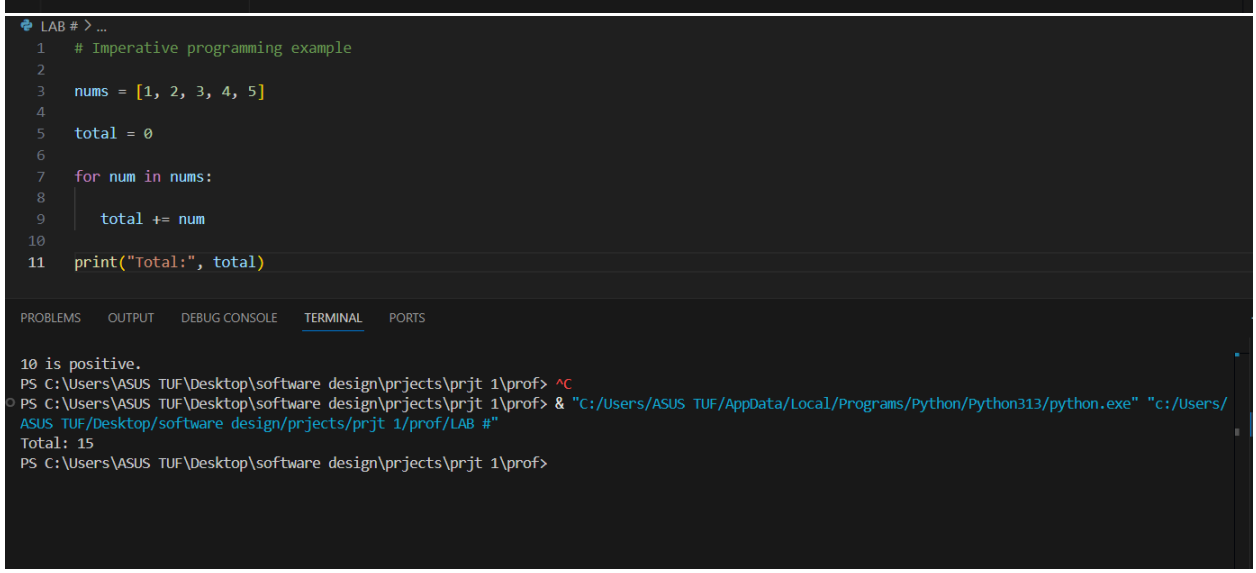
```
import tkinter as tk

def on_button_click():
  label.config(text="Button clicked!")

root = tk.Tk()
```

```
button = tk.Button(root, text="Click me!", command=on_button_click)
button.pack()
label = tk.Label(root, text="")
label.pack()
root.mainloop()
```

1. Discuss concurrency with the threading module.

   RESULT





```python
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age



    def greet(self):

        return f"Hello, my name is {self.name} and I am {self.age} years old."



p = Person("Alice", 25)

print(p.greet())
```

```
PS C:\Users\ASUS TUF\Desktop\software design\prjects\prjt 1\prof> & "C:/Users/ASUS TUF/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/ASUS TUF/Desktop/software design/prjects/prjt 1/prof/LAB #"
Hello, my name is Alice and I am 25 years old.
PS C:\Users\ASUS TUF\Desktop\software design\prjects\prjt 1\prof>
```

```python
nums = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x**2, nums))

even_nums = list(filter(lambda x: x % 2 == 0, nums))

print("Squared Numbers:", squared)

print("Even Numbers:", even_nums)
```

```
PS C:\Users\ASUS TUF\Desktop\software design\prjects\prjt 1\prof> & "C:/Users/ASUS TUF/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/ASUS TUF/Desktop/software design/prjects/prjt 1/prof/LAB #"
Squared Numbers: [1, 4, 9, 16, 25]
Even Numbers: [2, 4]
PS C:\Users\ASUS TUF\Desktop\software design\prjects\prjt 1\prof>
```

Follow-Up Questions

1. What are the key differences between imperative and declarative programming?

   ANS: Imperative programming focuses on *how* to perform tasks with explicit instructions and control flow, while declarative programming

focuses on *what* the desired outcome should be, leaving the system to determine how to achieve it.

2. In which scenarios would you prefer functional programming?

   ANS: In which scenarios would you prefer functional programming?

3. How can concurrency improve software performance?

   ANS: Concurrency can improve software performance by allowing multiple tasks to run simultaneously, making better use of system resources, reducing wait times, and improving overall throughput, especially in multi-core or distributed environments.