Lab 13: Code Refactoring – Improving Legacy Code with AI Suggestions

Course: AI Assisted Coding
Course Code: 23CS002PC304
Week: 7 – Friday
Regulation: R23
Program: B.Tech CSE

2303A52157

BATCH:34

Task 1: Refactoring – Removing Global Variables

◇ Legacy Code

```
rate = 0.1

def calculate_interest(amount):
  return amount * rate

print(calculate_interest(1000))
```

◇ Issues Identified

- Uses global variable `rate`
- Reduces modularity
- Hard to test and reuse

◇ Refactored Code

```
def calculate_interest(amount, rate):
  return amount * rate

print(calculate_interest(1000, 0.1))
```

◇ Improvements

- Removed global variable
- Increased modularity
- Easier unit testing

Task 2: Refactoring Deeply Nested Conditionals

◇ Legacy Code

```
score = 78

if score >= 90:
   print("Excellent")
else:
   if score >= 75:
      print("Very Good")
   else:
      if score >= 60:
         print("Good")
      else:
         print("Needs Improvement")
```

◇ Refactored Code (Flattened Logic)

```
score = 78

if score >= 90:
   print("Excellent")
elif score >= 75:
   print("Very Good")
elif score >= 60:
   print("Good")
else:
   print("Needs Improvement")
```

◇ Improvements

- Eliminated deep nesting
- Improved readability
- Easier maintenance

Task 3: Refactoring Repeated File Handling Code

◇ Legacy Code

```
f = open("data1.txt")
print(f.read())
f.close()

f = open("data2.txt")
print(f.read())
f.close()
```

◇ Refactored Code (Using Context Manager)

```
def read_file(filename):
    with open(filename, "r") as file:
        print(file.read())

read_file("data1.txt")
read_file("data2.txt")
```

◇ Improvements

- Used `with open()` (context manager)
- Avoided manual closing
- Followed DRY principle

Task 4: Optimizing Search Logic

◇ Legacy Code

```
users = ["admin", "guest", "editor", "viewer"]
name = input("Enter username: ")

found = False
for u in users:
  if u == name:
    found = True

print("Access Granted" if found else "Access Denied")
```

◇ Refactored Code (Using Set)

```
users = {"admin", "guest", "editor", "viewer"}  # O(1) lookup
name = input("Enter username: ")

print("Access Granted" if name in users else "Access Denied")
```

◇ Improvements

- Changed list → set
- Time Complexity improved:
    - List search: **O(n)**
    - Set lookup: **O(1)**
- Cleaner and faster

Task 5: Refactoring Procedural Code into OOP Design

◇ Legacy Code

```
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
```

◇ Refactored Code (OOP Design)

```
class EmployeeSalaryCalculator:
  def __init__(self, salary, tax_rate=0.2):
    self.salary = salary
    self.tax_rate = tax_rate

  def calculate_tax(self):
    return self.salary * self.tax_rate

  def calculate_net_salary(self):
    return self.salary - self.calculate_tax()

employee = EmployeeSalaryCalculator(50000)
print(employee.calculate_net_salary())
```

◇ Improvements

- Applied OOP principles
- Encapsulation of salary logic
- Reusable and scalable

Task 6: Refactoring for Performance Optimization

◇ Legacy Code

```
total = 0
for i in range(1, 1000000):
  if i % 2 == 0:
    total += i

print(total)
```

◇ Optimized Code (Mathematical Formula)

```
n = 999999
count = n // 2
total = count * (count + 1)
print(total)
```

◇ Explanation

Sum of first k even numbers = k(k + 1)

◇ Improvements

- Loop complexity: **O(n)**
- Formula complexity: **O(1)**
- Significant performance gain

Task 7: Removing Hidden Side Effects

◇ Legacy Code

```
data = []

def add_item(x):
  data.append(x)

add_item(10)
add_item(20)
print(data)
```

◇ Refactored Code (Functional Approach)

```
def add_item(data, x):
  return data + [x]

data = []
data = add_item(data, 10)
data = add_item(data, 20)

print(data)
```

◇ Improvements

- No mutation of global state
- Predictable behavior

- Easier testing

Task 8: Refactoring Complex Input Validation Logic

◇ Legacy Code

```
password = input("Enter password: ")

if len(password) >= 8:
  if any(c.isdigit() for c in password):
    if any(c.isupper() for c in password):
      print("Valid Password")
    else:
      print("Must contain uppercase")
  else:
    print("Must contain digit")
else:
  print("Password too short")
```

◇ Refactored Code (Modular Validation)

```
def is_long_enough(password):
  return len(password) >= 8

def has_digit(password):
  return any(c.isdigit() for c in password)

def has_uppercase(password):
  return any(c.isupper() for c in password)

def validate_password(password):
  if not is_long_enough(password):
    return "Password too short"
  if not has_digit(password):
    return "Must contain digit"
  if not has_uppercase(password):
    return "Must contain uppercase"
  return "Valid Password"

password = input("Enter password: ")
print(validate_password(password))
```

◇ Improvements

- Broke validation into small functions
- Improved readability
- High testability
- Follows Single Responsibility Principle
