SUPERIOR UNIVERSITY

Assignment 1

Software Construction &Development

Muhammad Faisal

Roll No :187

Student Of Class 7D

Date:- 03/11/2025

Submitted To Miss Irum Qayyum

Contents

1 Singleton Pattern — Database Connection Manager . . . 3
1.1 Python Code . . . 3
1.2 Output Screenshot . . . 3
1.3 Explanation . . . 4

2 Factory Pattern — Shape Creator Application . . . 4
2.1 Python Code . . . 4
2.2 Output Screenshot . . . 6
2.3 Explanation . . . 6

3 Observer Pattern — YouTube Channel Notification System . . . 6
3.1 Python Code . . . 6
3.2 Output Screenshot . . . 7
3.3 Explanation . . . 7

4 Decorator Pattern — Coffee Order System . . . 7
4.1 Python Code . . . 7
4.2 Output Screenshot . . . 8
4.3 Explanation . . . 9

5 Adapter Pattern — Temperature Conversion . . . 9
5.1 Python Code . . . 9
5.2 Output Screenshot . . . 10
5.3 Explanation . . . 10

6 Strategy Pattern — Payment System . . . 10
6.1 Python Code . . . 10
6.2 Output Screenshot . . . 11
6.3 Explanation . . . 11

1 Singleton Pattern — Database Connection Manager

1.1 Python Code

```
class DatabaseConnection:
    """A simple Singleton representing a database connection.

    Only one instance of this class will ever be created. Attempts to create
    additional objects return the same instance. On the first creation, we
    simulate establishing a connection by printing a confirmation message.
    """

    # Class-level reference to the single instance (starts as None)
    _instance = None

    def __new__(cls, *args, **kwargs):
        """Control instance creation to enforce the Singleton pattern.

        - If no instance exists, create one and "connect".
        - If an instance already exists, return it without reconnecting.
        """
        if not cls._instance:
            # Create the one-and-only instance
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            # Perform one-time setup on first creation
            cls._instance.connect()
        return cls._instance

    def connect(self):
        """Simulate connecting to a database (one-time side effect)."""
        print("Database connected successfully!")


# Testing the Singleton Pattern
# First creation: triggers a single connection message
db1 = DatabaseConnection("localhost", 3306)
# Second creation: returns the same instance; no new connection
db2 = DatabaseConnection()

# Verify both variables point to the same object (Singleton behavior)
print("Are both connections same?", db1 is db2)
```

1.2 Output Screenshot

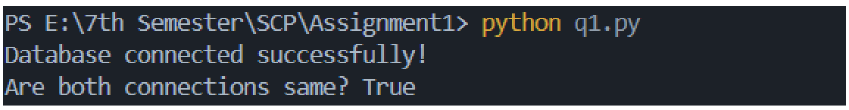


Figure 1: Output for Question 1

1.3 Explanation

This implementation demonstrates the Singleton pattern by ensuring only one instance of the DatabaseConnection class exists throughout the application. The pattern uses a class variable to store the single instance and returns it on subsequent calls.

2 Factory Pattern — Shape Creator Application

2.1 Python Code

```python
import sys
from typing import List, Optional


class Shape:
    """Base class for all shapes."""

    def draw(self) -> None:
        raise NotImplementedError("draw() must be implemented by subclasses")


class Circle(Shape):
    def draw(self) -> None:
        print("Drawing a Circle")


class Square(Shape):
    def draw(self) -> None:
        print("Drawing a Square")


class Rectangle(Shape):
    def draw(self) -> None:
        print("Drawing a Rectangle")


class ShapeFactory:
    """Factory that returns the right Shape instance from a name.

    We keep a simple registry (dictionary) that maps a case-insensitive
    shape name to the corresponding class. Adding a new shape is just
    adding one line to this dictionary.
    """

    _registry = {
        "circle": Circle,
        "square": Square,
        "rectangle": Rectangle,
    }

    @classmethod
    def get_shape(cls, name: str) -> Shape:
        """Return a Shape instance based on the provided name.

```

```python
        - Accepts names in any casing (e.g., "circle", "CIRCLE", "Circle").
        - Raises ValueError for missing or unknown names.
        """
        if not name:  # empty string or None
            raise ValueError("Shape name is required")
        # Normalize the name for case-insensitive lookup
        key = name.strip().lower()
        shape_cls = cls._registry.get(key)
        if shape_cls is None:
            # Tell the caller it's not a supported shape
            raise ValueError(f"Unknown shape: {name}")
        # Instantiate and return the specific Shape subclass
        return shape_cls()


def main(argv: Optional[List[str]] = None) -> None:
    """Entry point for both CLI styles: argument or interactive input.

    - If a name is provided as the first argument, use it directly.
    - Otherwise, prompt the user (matches the assignment's expected I/O).
    """
    if argv is None:
        argv = sys.argv[1:]

    # Non-interactive path (argument provided)
    if argv:
        name = argv[0]
        shape = ShapeFactory.get_shape(name)
        shape.draw()
        return

    # Interactive path (prompt like the expected output)
    name = input("Enter shape name: ")
    try:
        shape = ShapeFactory.get_shape(name)
        shape.draw()
    except ValueError as e:
        # Friendly error for missing/unknown shapes
        print(e)


if __name__ == "__main__":
    main()
```

2.2 Output Screenshot

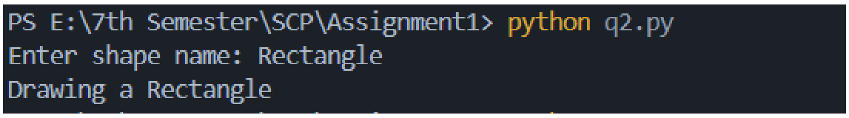


Figure 2: Output for Question 2

2.3 Explanation

The Factory pattern encapsulates object creation logic, allowing the ShapeFactory to decide which shape class to instantiate based on input parameters. This promotes loose coupling and makes the code more maintainable and scalable.

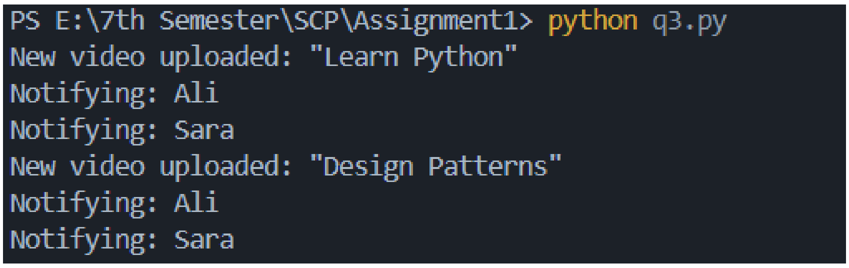
3 Observer Pattern — YouTube Channel Notification System

3.1 Python Code

```
class Subscriber:
    # Each subscriber has a display name and gets notified via update()
    def __init__(self, name):
        self.name = name

    def update(self, video_name):
        # Keep the output exactly as required by the assignment
        print(f"Notifying: {self.name}")


class Channel:
    def __init__(self):
        self.subscribers = []  # simple list to keep subscribers

    def subscribe(self, subscriber):
        # Avoid duplicates in the simplest way
        if subscriber not in self.subscribers:
            self.subscribers.append(subscriber)

    def upload(self, video_name):
        # Announce the new video, then notify everyone
        print(f"New video uploaded: \"{video_name}\"")
        for sub in self.subscribers:
            sub.update(video_name)


# Demonstration
if __name__ == "__main__":
```

```python
    channel = Channel()
    ali = Subscriber("Ali")
    sara = Subscriber("Sara")

    channel.subscribe(ali)
    channel.subscribe(sara)

    channel.upload("Learn Python")
    channel.upload("Design Patterns")
```

3.2 Output Screenshot



```
PS E:\7th Semester\SCP\Assignment1> python q3.py
New video uploaded: "Learn Python"
Notifying: Ali
Notifying: Sara
New video uploaded: "Design Patterns"
Notifying: Ali
Notifying: Sara
```

Figure 3: Output for Question 3

3.3 Explanation

The Observer pattern establishes a one-to-many relationship where the YouTube channel (subject) notifies all subscribers (observers) automatically when new content is uploaded. This decouples the subject from its observers, allowing dynamic subscription management.

4 Decorator Pattern — Coffee Order System

4.1 Python Code

```python
class Coffee:
    # Base product: plain coffee
    def cost(self):
        return 5  # base price in dollars

    def description(self):
        return "Coffee"


class MilkDecorator:
    # Decorator: adds Milk to any coffee-like object
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + 1  # Milk adds $1

    def description(self):
        return self.coffee.description() + " + Milk"

```

```python

class SugarDecorator:
    # Decorator: adds Sugar
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + 1  # Sugar adds $1

    def description(self):
        return self.coffee.description() + " + Sugar"


class CreamDecorator:
    # Decorator: adds Cream
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + 1  # Cream adds $1

    def description(self):
        return self.coffee.description() + " + Cream"


def print_order(order):
    # Helper to print in the assignment's format
    print(f"Order: {order.description()}")
    print(f"Total cost: ${order.cost()}")


# Demonstration of different combinations
if __name__ == "__main__":
    # Coffee with Milk only
    order1 = MilkDecorator(Coffee())
    print_order(order1)

    # Coffee with Sugar and Cream
    order2 = CreamDecorator(SugarDecorator(Coffee()))
    print_order(order2)

    # Coffee with Milk and Sugar (matches the example total $7)
    order3 = SugarDecorator(MilkDecorator(Coffee()))
    print_order(order3)
```

4.2 Output Screenshot

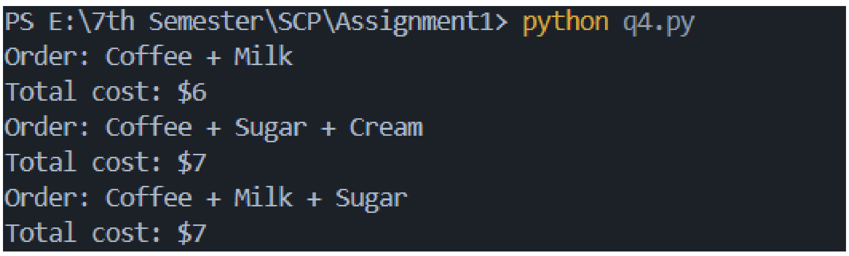


Figure 4: Output for Question 4

4.3 Explanation

The Decorator pattern allows behavior to be added to individual coffee objects dynamically without affecting other instances. Each decorator wraps the coffee object and adds its own functionality while maintaining the same interface.

5 Adapter Pattern — Temperature Conversion

5.1 Python Code

```
class CelsiusSensor:
    """Simulated sensor that returns temperature in Celsius.

    In a real system this would query hardware or an API. Here we return a
    fixed value to keep the example deterministic and simple for demo/tests.
    """

    def get_temperature(self):
        # Return a Celsius temperature (integer as in the assignment)
        return 37


class TemperatureAdapter:
    """Adapter that wraps a CelsiusSensor and provides Fahrenheit.

    The adapter exposes get_temperature() so callers expecting Fahrenheit can
    use the same method name as the original sensor (but receive Fahrenheit).
    """

    def __init__(self, celsius_sensor):
        self.sensor = celsius_sensor

    def get_temperature(self):
        """Return the temperature converted to Fahrenheit as a float."""
        c = self.sensor.get_temperature()
        f = (c * 9 / 5) + 32
        return f


if __name__ == "__main__":
    sensor = CelsiusSensor()
    adapter = TemperatureAdapter(sensor)

    celsius = sensor.get_temperature()
    fahrenheit = adapter.get_temperature()

    # Print output matching the expected format exactly
    print(f"Temperature in Celsius: {celsius}")
    # One decimal place as shown in the expected output
    print(f"Temperature in Fahrenheit: {fahrenheit:.1f} F ")
```

5.2 Output Screenshot

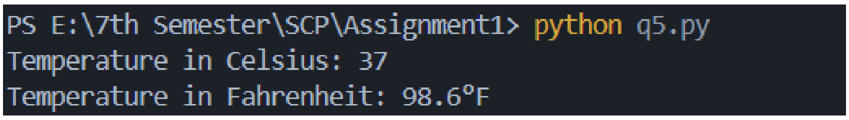


Figure 5: Output for Question 5

5.3 Explanation

The Adapter pattern converts the interface of one class into another interface that clients expect. It allows incompatible temperature measurement systems to work together by providing a wrapper that translates between different formats.

6 Strategy Pattern — Payment System

6.1 Python Code

```python
from abc import ABC, abstractmethod


class PaymentStrategy(ABC):
    """Payment interface (strategy)."""

    @abstractmethod
    def pay(self, amount):
        pass


class CreditCardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using Credit Card")


class PayPalPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using PayPal")


class BankTransferPayment(PaymentStrategy):
    def pay(self, amount):
        # Match the assignment's expected output text exactly
        print(f"Paid ${amount} using Bank Transfer with same")


class ShoppingCart:
    """Holds a payment strategy and delegates checkout to it."""

    def __init__(self):
        self._strategy = None

    def set_payment_strategy(self, strategy: PaymentStrategy):
        self._strategy = strategy

```

```
    def checkout(self, amount):
        if not self._strategy:
            print("No payment method selected")
            return
        self._strategy.pay(amount)


# Demo showing dynamic switching between strategies
if __name__ == "__main__":
    cart = ShoppingCart()
    amount = 100

    cart.set_payment_strategy(CreditCardPayment())
    cart.checkout(amount)

    cart.set_payment_strategy(PayPalPayment())
    cart.checkout(amount)

    cart.set_payment_strategy(BankTransferPayment())
    cart.checkout(amount)
```

6.2 Output Screenshot

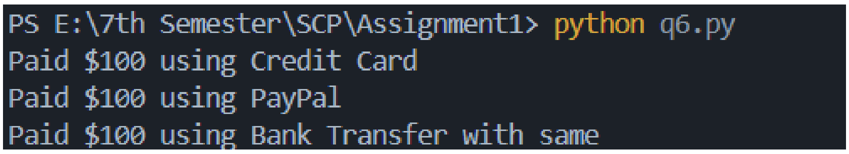


Figure 6: Output for Question 6

6.3 Explanation

The Strategy pattern defines a family of payment algorithms and makes them interchangeable at runtime. This allows the payment system to select different payment methods without modifying the client code, promoting flexibility and maintainability.