**Behavioral Design Patterns**

**1. Observer Pattern: Weather Station**

Use Case: A weather station that updates multiple displays (e.g., temperature, humidity, pressure) in real-time when the weather data changes.

In this scenario, the weather station is the subject, and the displays are the observers. When the weather data changes, the weather station notifies all the displays to update their values.

**CODE**

class WeatherStation:

def \_\_init\_\_(self):

self.\_observers = []

self.\_temperature = 0

self.\_humidity = 0

self.\_pressure = 0

def add\_observer(self, observer):

self.\_observers.append(observer)

def notify\_observers(self):

for observer in self.\_observers:

observer.update(self.\_temperature, self.\_humidity, self.\_pressure)

def update\_measurements(self, temperature, humidity, pressure):

self.\_temperature = temperature

self.\_humidity = humidity

self.\_pressure = pressure

self.notify\_observers()

class Display:

def update(self, temperature, humidity, pressure):

pass

class TemperatureDisplay(Display):

def update(self, temperature, humidity, pressure):

print(f"Temperature: {temperature}°C")

class HumidityDisplay(Display):

def update(self, temperature, humidity, pressure):

print(f"Humidity: {humidity}%")

**2. Strategy Pattern: Payment Gateway**

Use Case: A payment gateway that allows users to pay using different payment methods (e.g., credit card, PayPal, bank transfer).

In this scenario, the payment gateway is the context, and the payment methods are the strategies. The payment gateway delegates the payment processing to the selected payment method.

**CODE:**

class PaymentGateway:

def \_\_init\_\_(self):

self.\_payment\_method = None

def set\_payment\_method(self, payment\_method):

self.\_payment\_method = payment\_method

def process\_payment(self, amount):

if self.\_payment\_method:

self.\_payment\_method.pay(amount)

else:

print("No payment method selected")

class PaymentMethod:

def pay(self, amount):

pass

class CreditCardPayment(PaymentMethod):

def pay(self, amount):

print(f"Paid {amount} using credit card")

class PayPalPayment(PaymentMethod):

def pay(self, amount):

print(f"Paid {amount} using PayPal")

**Creational Design Patterns**

**1. Factory Pattern: Vehicle Factory**

Use Case: A vehicle factory that produces different types of vehicles (e.g., car, truck, motorcycle).

In this scenario, the vehicle factory is the factory, and the vehicles are the products. The factory creates instances of the appropriate vehicle type based on the input parameters.

**CODE:**

class VehicleFactory:

def create\_vehicle(self, vehicle\_type):

if vehicle\_type == "car":

return Car()

elif vehicle\_type == "truck":

return Truck()

elif vehicle\_type == "motorcycle":

return Motorcycle()

else:

return None

class Vehicle:

def drive(self):

pass

class Car(Vehicle):

def drive(self):

print("Driving a car")

class Truck(Vehicle):

def drive(self):

print("Driving a truck")

class Motorcycle(Vehicle):

def drive(self):

print("Driving a motorcycle")

**2. Singleton Pattern: Logger**

Use Case: A logger that logs messages to a file, ensuring that only one instance of the logger exists throughout the application.

In this scenario, the logger is the singleton, and it ensures that only one instance is created and reused throughout the application.

**CODE:**

class Logger:

\_instance = None

\_log\_file = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(Logger, cls).\_\_new\_\_(cls)

cls.\_log\_file = open("log.txt", "w")

return cls.\_instance

def log(self, message):

self.\_log\_file.write(message + "\n")

self.\_log\_file.flush()

**Structural Design Patterns**

**1. Adapter Pattern: File Converter**

Use Case: A file converter that converts files from one format to another (e.g., CSV to JSON).

In this scenario, the file converter is the adapter, and it adapts the file format from one type to another.

class JSONConverter:

def convert(self, csv\_content):

pass

class CSVParser:

def parse(self, csv\_content):

return "{\"key\":\"value\"}"

class CSVToJSONAdapter(JSONConverter):

def \_\_init\_\_(self, csv\_parser):

self.\_csv\_parser = csv\_parser

def convert(self, csv\_content):

return self.\_csv\_parser.parse(csv\_content)

**2. Bridge Pattern: Drawing Application**

Use Case: A drawing application that allows users to draw different shapes (e.g., circle, rectangle) using different drawing tools (e.g., pencil, brush).

In this scenario, the drawing application is the abstraction, and the shapes and drawing tools are the implementations. The bridge pattern decouples the abstraction from the implementation, allowing for more flexibility and extensibility.

**CODE:**

class DrawingApplication:

def \_\_init\_\_(self, drawing\_tool):

self.\_drawing\_tool = drawing\_tool

def draw\_shape(self, shape):

self.\_drawing\_tool.draw(shape)

class Shape:

def draw(self):

pass

class Circle(Shape):

def draw(self):

print("Drawing a circle")

class Rectangle(Shape):

def draw(self):

print("Drawing a rectangle")

class DrawingTool:

def draw(self, shape):

pass

class Pencil(DrawingTool):

def draw(self, shape):

print("Drawing with a pencil")

shape.draw()

class Brush(DrawingTool):

def draw(self, shape):

print("Drawing with a brush")

shape.draw()