Exercise 1: Problem Statement on Design patterns Come up creatively with six different use cases to demonstrate your understanding of the following software design patterns by coding the same.

1. Two use cases to demonstrate two behavioural design pattern.
2. Two use cases to demonstrate two creational design pattern.
3. Two use cases to demonstrate two structural design pattern.

#1.1

# Observer Pattern

class WeatherStation:

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

self.\_observers = []

self.\_temperature = None

def register\_observer(self, observer):

self.\_observers.append(observer)

def remove\_observer(self, observer):

self.\_observers.remove(observer)

def notify\_observers(self):

for observer in self.\_observers:

observer.update(self.\_temperature)

def set\_temperature(self, temperature):

self.\_temperature = temperature

self.notify\_observers()

class Observer:

def update(self, temperature):

pass

class PhoneDisplay(Observer):

def update(self, temperature):

print(f"Phone display: Temperature updated to {temperature}°C")

class TabletDisplay(Observer):

def update(self, temperature):

print(f"Tablet display: Temperature updated to {temperature}°C")

class DesktopDisplay(Observer):

def update(self, temperature):

print(f"Desktop display: Temperature updated to {temperature}°C")

# Usage

weather\_station = WeatherStation()

phone\_display = PhoneDisplay()

tablet\_display = TabletDisplay()

desktop\_display = DesktopDisplay()

weather\_station.register\_observer(phone\_display)

weather\_station.register\_observer(tablet\_display)

weather\_station.register\_observer(desktop\_display)

weather\_station.set\_temperature(25)

weather\_station.set\_temperature(30)

#1.2

# Strategy Pattern

class PaymentStrategy:

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 BitcoinPayment(PaymentStrategy):

def pay(self, amount):

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

class PaymentContext:

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

self.\_strategy = strategy

def set\_strategy(self, strategy):

self.\_strategy = strategy

def pay(self, amount):

self.\_strategy.pay(amount)

# Usage

context = PaymentContext(CreditCardPayment())

context.pay(100)

context.set\_strategy(PayPalPayment())

context.pay(200)

context.set\_strategy(BitcoinPayment())

context.pay(300)

#2.1

# Factory Method Pattern

class Document:

def create(self):

pass

class WordDocument(Document):

def create(self):

print("Word document created")

class ExcelDocument(Document):

def create(self):

print("Excel document created")

class PDFDocument(Document):

def create(self):

print("PDF document created")

class DocumentFactory:

def create\_document(self, doc\_type):

if doc\_type == 'word':

return WordDocument()

elif doc\_type == 'excel':

return ExcelDocument()

elif doc\_type == 'pdf':

return PDFDocument()

else:

raise ValueError("Unknown document type")

# Usage

factory = DocumentFactory()

doc = factory.create\_document('word')

doc.create()

doc = factory.create\_document('excel')

doc.create()

doc = factory.create\_document('pdf')

doc.create()

#2.2

#command pattern

class Command:

def execute(self):

raise NotImplementedError

class LightOnCommand(Command):

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

self.\_light = light

def execute(self):

self.\_light.turn\_on()

class LightOffCommand(Command):

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

self.\_light = light

def execute(self):

self.\_light.turn\_off()

class Light:

def turn\_on(self):

print("The light is on")

def turn\_off(self):

print("The light is off")

class RemoteControl:

def set\_command(self, command):

self.\_command = command

def press\_button(self):

self.\_command.execute()

# Usage

light = Light()

light\_on\_command = LightOnCommand(light)

light\_off\_command = LightOffCommand(light)

remote = RemoteControl()

remote.set\_command(light\_on\_command)

remote.press\_button()

remote.set\_command(light\_off\_command)

remote.press\_button()

#3.1

#singleton pattern

class Logger:

\_instance = None

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

if cls.\_instance is None:

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

return cls.\_instance

def log(self, message):

print(f"Log: {message}")

# Usage

logger1 = Logger()

logger2 = Logger()

logger1.log("This is the first log message.")

logger2.log("This is the second log message.")

print(logger1 is logger2) # Output: True

#3.2

#adapter pattern

class OldPaymentSystem:

def make\_payment(self, amount):

print(f"Payment of {amount} made using Old Payment System")

class NewPaymentService:

def process\_payment(self, amount):

print(f"Payment of {amount} processed using New Payment Service")

class PaymentAdapter:

def \_\_init\_\_(self, new\_payment\_service):

self.\_new\_payment\_service = new\_payment\_service

def make\_payment(self, amount):

self.\_new\_payment\_service.process\_payment(amount)

# Usage

old\_payment\_system = OldPaymentSystem()

old\_payment\_system.make\_payment(100)

new\_payment\_service = NewPaymentService()

adapter = PaymentAdapter(new\_payment\_service)

adapter.make\_payment(200)