Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJ19ITC802 **DATE:** 25-03-2024

COURSE NAME: Design Patterns Laboratory **CLASS:** BE - IT

EXPERIMENT NO. 8

CO/LO: Identify and apply the most suitable design pattern to address a given application design problem.

AIM: Implement the Multiple Pattern using any language of your choice for any real-life scenario.

DESCRIPTION:

Combining multiple design patterns is a common practice in software development. This approach allows developers to solve complex problems by using a combination of different patterns, each addressing a specific aspect of the problem. For instance, in a system where a family of sensors is manufactured by different factories, the Observer pattern can be used to notify when a sensor is triggered, and the Abstract Factory pattern can be used to create the sensors.

SOURCE CODE:

```
from abc import ABC, abstractmethod
from enum import Enum

# Factory Pattern: Create shipments of different types [Air, Sea, Land]

class ShipmentType(Enum):
    AIR = "Air"
    SEA = "Sea"
    LAND = "Land"

class Shipment(ABC):
    @abstractmethod
    def deliver(self):
```

```
        pass

class AirShipment(Shipment):
    def deliver(self):
        print("Delivering by Air")

class SeaShipment(Shipment):
    def deliver(self):
        print("Delivering by Sea")

class LandShipment(Shipment):
    def deliver(self):
        print("Delivering by Land")

class ShipmentFactory:
    def create_shipment(self, shipment_type):
        if shipment_type == ShipmentType.AIR:
            return AirShipment()
        elif shipment_type == ShipmentType.SEA:
            return SeaShipment()
        elif shipment_type == ShipmentType.LAND:
            return LandShipment()
        else:
            raise ValueError("Invalid shipment type")

# Proxy Pattern: Control access to sensitive shipment data

class ShipmentProxy(Shipment):
    def __init__(self, real_shipment):
        self._real_shipment = real_shipment

    def deliver(self):
        if self.check_access():
```

```
            self._real_shipment.deliver()
        else:
            print("Access denied")

    def check_access(self):
        # Simulated access control logic
        return True

# Observer Pattern: Notify the logistics manager whenever the status of a shipment changes

class ShipmentSubject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, shipment):
        for observer in self._observers:
            observer.update(shipment)

class ShipmentObserver:
    def __init__(self, name):
        self._name = name

    def update(self, shipment):
        print(f"Logistics Manager {self._name}: Shipment status changed to {shipment}")

# Usage
```

```
if __name__ == "__main__":
    # Factory Pattern: Create shipments of different types
    factory = ShipmentFactory()
    air_shipment = factory.create_shipment(ShipmentType.AIR)
    sea_shipment = factory.create_shipment(ShipmentType.SEA)
    land_shipment = factory.create_shipment(ShipmentType.LAND)

    # Proxy Pattern: Control access to sensitive shipment data
    proxy = ShipmentProxy(air_shipment)
    proxy.deliver()

    # Observer Pattern: Notify the logistics manager whenever the status of a shipment
changes
    subject = ShipmentSubject()
    logistics_manager1 = ShipmentObserver("1")
    logistics_manager2 = ShipmentObserver("2")
    subject.attach(logistics_manager1)
    subject.attach(logistics_manager2)

    # Simulate a status change and notify observers
    subject.notify_observers("In transit")

      # Proxy Pattern: Control access to sensitive shipment data
    proxy1 = ShipmentProxy(sea_shipment)
    proxy1.deliver()

    # Observer Pattern: Notify the logistics manager whenever the status of a shipment
changes
    subject1 = ShipmentSubject()
    logistics_manager3 = ShipmentObserver("3")
    logistics_manager4 = ShipmentObserver("4")
    subject1.attach(logistics_manager1)
    subject1.attach(logistics_manager3)
```

```
subject1.attach(logistics_manager4)
# Simulate a status change and notify observers
subject1.notify_observers("In transit")
subject1.notify_observers("Shipped")
```

OUTPUT

Delivering by Air
Logistics Manager 1: Shipment status changed to In transit
Logistics Manager 2: Shipment status changed to In transit
Delivering by Sea
Logistics Manager 1: Shipment status changed to In transit
Logistics Manager 3: Shipment status changed to In transit
Logistics Manager 4: Shipment status changed to In transit
Logistics Manager 1: Shipment status changed to Shipped
Logistics Manager 3: Shipment status changed to Shipped
Logistics Manager 4: Shipment status changed to Shipped

CONCLUSION:

In This Experiment, we integrate the Factory, Proxy, and Observer design patterns to facilitate the monitoring and management of shipments within a logistics company. The Factory Pattern is utilized to create shipments of various types (Air, Sea, Land) through the **ShipmentFactory** class, while the Proxy Pattern ensures controlled access to sensitive shipment data via the **ShipmentProxy** class. The Observer Pattern is employed to notify logistics managers of shipment status changes, with the **ShipmentSubject** class managing observer attachment and notification, and **ShipmentObserver** classes representing individual managers. Overall, this system provides a structured approach to handling different aspects of shipment management, from creation to access control and real-time status updates.

REFERENCES:

[1] https://www.geeksforgeeks.org/singleton-design-pattern-introduction/
[2] https://www.geeksforgeeks.org/factory-method-for-designing-pattern/