SDA LAB 01

Objective: To implement design patterns (Singleton, Strategy, Factory, Observer)

Theory:

Singleton:

- This design pattern ensures that only one instance of a class exists throughout the application.
- It's useful for things like logging, configuration, or database connections where you need a single point of control.
- Key features include a private constructor (to prevent direct instantiation), a static method to access the single instance, and a way to initialize the instance.

Strategy:

- This pattern lets you define a set of algorithms, each encapsulated in its own class.
- You can then choose and swap out these algorithms at runtime.
- Key elements are an interface for the algorithms, concrete classes implementing that interface, and a "context" class that uses one of the algorithms.

Factory:

- This pattern handles the creation of objects, making the client code less dependent on the specific types of objects being created.
- A "factory" class takes on the responsibility of creating objects, promoting flexibility and easier maintenance.

Observer:

- This pattern establishes a one-to-many relationship between objects.
- When one object (the "subject") changes, all objects that are "observing" it are automatically notified and updated.
- This is common in event-driven systems, and key aspects include loose coupling between the subject and its observers.

Observation:

1. singletonPattern.py

```
class ApplicationState:

    instance=None

    def __init__(self):

        self.isLoggedIn=False

    @staticmethod

    def getAppState():

        if not ApplicationState.instance:

            ApplicationState.instance=ApplicationState()

            return ApplicationState.instance

appState1= ApplicationState.getAppState()

print(appState1.isLoggedIn)  #False


appState2= ApplicationState.getAppState()

appState1.isLoggedIn=True


print(appState1.isLoggedIn) #false

print(appState2.isLoggedIn) #true
```

2. strategyPattern.py

```
from abc import ABC, abstractmethod


# Abstract base class for filter strategies

class FilterStrategy(ABC):

    # Abstract method to be implemented by concrete strategies
```

```
    @abstractmethod
    def removeValue(self, val):
        pass
# Concrete strategy to remove negative values
class RemoveNegativeStrategy(FilterStrategy):
    # Returns True if the value is negative, indicating it should be removed
    def removeValue(self, val):
        return val < 0
# Concrete strategy to remove odd values
class RemoveOddStrategy(FilterStrategy):
    # Returns True if the value is odd, indicating it should be removed
    def removeValue(self, val):
        return abs(val) % 2 == 1
# Class to hold and filter a list of values
class Values:
    def __init__(self, vals):
        self.vals = vals  # Initialize with a list of values
    # Filters the values using a given filter strategy
    def filter(self, strategy):
        res = []  # Initialize an empty list for filtered results
        for n in self.vals:  # Iterate through each value
            # If the strategy doesn't recommend removal, keep the value
            if not strategy.removeValue(n):
                res.append(n)
```

```
        return res  # Return the filtered list

# Create a Values instance with a list of integers
values = Values([-7, -4, -1, 0, 2, 6, 9])

# Apply the RemoveNegativeStrategy and print the result
print(values.filter(RemoveNegativeStrategy()))  # Output: [0, 2, 6, 9]

# Apply the RemoveOddStrategy and print the result
print(values.filter(RemoveOddStrategy()))  # Output: [-4, 0, 2, 6]
```

3. factoryPattern.py

```
class Burger:

    def __init__(self,ingredients):

        self.ingredients=ingredients

    def print(self):

        print(self.ingredients)

class BurgerFactory:

    def createCheeseBurger(self):

        ingredients = ["bhakku pauroti","cheese","patty "]

        return Burger(ingredients)

    def createDeluxeCheeseBurger(self):

        ingredients = ["bhakku pauroti","cheese","patty
","tomato","lettuce"]

        return Burger(ingredients)

    def createVeganBurger(self):

        ingredients= ["bhakku pauroti","special-sauce","patty "]

        return Burger(ingredients)

burgerFactory= BurgerFactory()
```

```
burgerFactory.createCheeseBurger().print()

burgerFactory.createDeluxeCheeseBurger().print()

burgerFactory.createVeganBurger().print()
```

4. observerPattern.py

```
from abc import ABC, abstractmethod
# Subject class (YoutubeChannel)
class YoutubeChannel:

    def __init__(self, name):

        self.name = name

        self.subscribers = []


    def subscribe(self, sub):

        self.subscribers.append(sub)



    def notify(self, event):

        for sub in self.subscribers:

            sub.sendNotification(self.name, event)
# Abstract Observer class (YoutubeSubscriber)

class YoutubeSubscriber(ABC):

    @abstractmethod

    def sendNotification(self, channel_name, event):

        pass

# Concrete Observer class (YoutubeUser)

class YoutubeUser(YoutubeSubscriber):

    def __init__(self, name):

        self.name = name
```

```
    def sendNotification(self, channel_name, event):

        print(f"{self.name} received notification from {channel_name}:
{event}")


# Main Program

channel = YoutubeChannel("Dami Channel")

# Creating and subscribing users

channel.subscribe(YoutubeUser("HeroKancha1"))

channel.subscribe(YoutubeUser("RamriKanxi2"))

channel.subscribe(YoutubeUser("PapakiPari3"))

# Notifying all subscribers
channel.notify("Naya video aaisakeko cha herdim hai !")
```

Results :

- **Singleton Pattern**: Confirmed only one instance is created.
- **Strategy Pattern**: Successfully switched between algorithms.
- **Factory Pattern**: Created objects dynamically based on input.
- **Observer Pattern**: Observers received updates efficiently

Conclusion :

Design patterns provide reusable solutions to common software design challenges. Implementing Singleton, Strategy, Factory, and Observer patterns demonstrates their utility in real-world applications.