Design Pattern in OOPs

(A) Singleton Pattern
—----------------------------
For Example:
==========
```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)
```

========== Explanation of Example ====================

`_instance` is a class variable.

It will store the only instance of this class.
`__new__` is responsible for creating a new object (before `__init__`).

`cls._instance is None`: Checks if the class has created an object before.
cls._instance = super().__new__(cls) // Creates a new object using the parent class (`object`) and stores it in `_instance`.

If an instance already exists**:**

It returns the old one instead of creating a new object.

obj1 = Singleton()

obj2 = Singleton()

Even though we call `Singleton()` two times, the Singleton pattern returns the same object both times.

Summary

`Singleton()` called multiple times → returns the same object

`_instance` stores the single instance

`__new__` ensures only one object is created

`obj1 is obj2` → **True**

Real-Life Use Cases of Singleton

1. Database Connection

Only one connection object should exist to avoid multiple unnecessary DB connections.

2. Logger System

You want all parts of your application to log using a single common logger.

3. Configuration Manager

App settings (API keys, environment variables, paths) should be loaded once.

4. Cache or Session Manager

You keep shared data in a single place across the whole app.

5. Thread Pool Manager

Creating multiple thread pools is costly—Singleton ensures only one exists.

(B) Factory Pattern

For Example:

```
class Shape:
    def draw(self):
        pass
```

```
class Circle(Shape):
    def draw(self):
        print("Drawing Circle")

class Square(Shape):
    def draw(self):
        print("Drawing Square")

class ShapeFactory:
    def get_shape(self, shape_type):
        if shape_type == "circle":
            return Circle()
        elif shape_type == "square":
            return Square()

factory = ShapeFactory()
shape = factory.get_shape("circle")
shape.draw()
```

==========explanation of The code==========

```
class Shape:
    def draw(self):
        pass
```

This is a base class.
`draw()` does nothing (`pass`).
Child classes will override this method.

```
class Circle(Shape):
    def draw(self):
        print("Drawing Circle")
```

Circle inherits from Shape.
Circle has its own version of draw().
This is Polymorphism (same method, different output).

```
class Square(Shape):
    def draw(self):
        print("Drawing Square")
```

Same as Circle but prints Square.
Again: method overriding.

Factory Class – Creates Objects

```
class ShapeFactory:
    def get_shape(self, shape_type):
        if shape_type == "circle":
            return Circle()
        elif shape_type == "square":
            return Square()
```

What this does?

Instead of writing:

```
shape = Circle()
```

You write:

```
shape = factory.get_shape("circle")
```

Factory Pattern: You don't create objects directly.

The factory decides which object to create.

Using the Factory

```
factory = ShapeFactory()
shape = factory.get_shape("circle")
shape.draw()
```

Step-by-step:

1. Create factory
   → ShapeFactory()
2. Ask the factory for a shape
   → "circle" → it returns a Circle object
3. Call draw()
   → prints Drawing Circle

Real-Life Use Cases of Factory

Real-Life System	Factory Creates
Payment System	bKash, Nagad, Visa, PayPal
Notification System	Email, SMS, Push
DB Connections	MySQL, SQLite, PostgreSQL
UI Widgets	Button, Checkbox, Input
Games	Enemy Object
Operating System	Drivers
ML Apps	Different Modules
Cloud Services	Storage types

(C) Builder Pattern

For Example :

```
class BurgerBuilder:
    def __init__(self):
        self.burger = {}

    def add_bread(self):
        self.burger['bread'] = 'sesame'
        return self

    def add_meat(self):
        self.burger['meat'] = 'chicken'
        return self

    def build(self):
        return self.burger

burger = BurgerBuilder().add_bread().add_meat().build()
print(burger)
```

Class Definition

```
class BurgerBuilder:
```

You are creating a class that will build a burger.

2️⃣ **C**onstructor

```
def __init__(self):
    self.burger = {}
```

- When the object is created, an empty dictionary is made.
- This dictionary will store burger parts like:
    - bread
    - meat
    - cheese
    - etc.

Example:
`{'bread': 'sesame', 'meat': 'chicken'}`

3️⃣ Add Bread Method

```
def add_bread(self):
    self.burger['bread'] = 'sesame'
    return self
```

- Adds `'bread': 'sesame'` to the burger.
- Returns self (very important)

Returning `self` allows method chaining:

```
BurgerBuilder().add_bread().add_meat()
```

4️⃣ Add Meat Method

```
def add_meat(self):
    self.burger['meat'] = 'chicken'
    return self
```

- Adds `'meat': 'chicken'`
- Returns `self` to keep chaining.

5️⃣ Build Method

```
def build(self):
    return self.burger
```

- Returns the final burger as a dictionary.
- This is the final result of the builder.

Using the Builder

```
burger = BurgerBuilder().add_bread().add_meat().build()
```

Let's break it:

1. `BurgerBuilder()` → create builder
2. `.add_bread()` → add bread
3. `.add_meat()` → add meat
4. `.build()` → return final burger

Real-Life Software Use Cases of the Builder Pattern

1. Building SQL Queries (Most Common Use Case)
2. Building API Requests

3. Building UI Components (Flutter, React, Tkinter)
4. Game Development (Unity / Unreal Engine)
5. Object Serialisation / JSON Building
6. Building Emails (Email Clients)
7. Cloud Infrastructure (Terraform, AWS CDK)
8. Machine Learning Model Pipelines
9. Building Web Pages (HTML Builders)
10. Creating Config Files

(D) Adapter Pattern Concept

The Adapter Pattern allows two incompatible interfaces to work together.

One class has an interface your client expects.

Another class has a different interface

The adapter translates one interface into another.

=====Explanation of code=== for Example===

```
class EuropeanPlug:
    def connect_eu(self):
        print("Using European plug")

class USAdapter:
    def __init__(self, plug):
        self.plug = plug

    def connect_us(self):
        self.plug.connect_eu()

plug = EuropeanPlug()
adapter = USAdapter(plug)
```

Real-Life Analogy

Imagine you bought an EU device, but your home has US sockets.

- EuropeanPlug = device plug
- USAdapter = the adapter you plug into a US socket

- connect_us() = you can now use your EU device in the US

Key Points

1. Adapter implements the interface expected by the client.
2. Adapter holds a reference to the original class.
3. Client code can call expected methods without worrying about incompatible interfaces.

Real-Life Software Use Cases of the Adapter Pattern

Adapter Pattern in Real Software

The Adapter Pattern is used whenever you need to make two incompatible interfaces work together. Here are some concrete examples:

---

1. Payment Gateways

- You have an e-commerce system that expects a standard payment interface (`pay(amount)`), but you want to use multiple payment providers like PayPal, Stripe, or local banks.
- Each provider has a different API.
- Adapter wraps each provider so your system can call the same `pay()` method for all of them.

Example:

```
class PayPal:
    def send_payment(self, amount):
        print(f"PayPal payment: {amount}")

class Stripe:
    def make_payment(self, amount):
        print(f"Stripe payment: {amount}")

class PayPalAdapter:
    def __init__(self, paypal):
        self.paypal = paypal
    def pay(self, amount):
```

```
        self.paypal.send_payment(amount)

class StripeAdapter:
    def __init__(self, stripe):
        self.stripe = stripe
    def pay(self, amount):
        self.stripe.make_payment(amount)

# Client code uses same interface
payments = [PayPalAdapter(PayPal()), StripeAdapter(Stripe())]
for payment in payments:
    payment.pay(100)
```

2. Legacy Code Integration

Suppose your system has a new logging system, but some old modules use a different logging API.

The adapter allows old modules to log using the new system without changing old code.

3. File Format Conversions

Example: A program expects JSON input, but some clients provide XML files

An Adapter converts XML → JSON so the program can use the data seamlessly.

4. UI Components

Suppose you have a GUI library, and you want to **reuse custom components** from another library with a different interface.
Adapter allows them to work without rewriting the components.

Key Idea

Adapter is mostly used for integration.

It saves time, avoids changing existing code, and allows code reuse.

(E) Decorator Pattern

—----------

Adds new features to an object without modifying its structure.

```
def make_uppercase(func):
    def wrapper():
        return func().upper()
    return wrapper

@make_uppercase
def say_hello():
    return "hello world"

print(say_hello())
```

==========Explain the code==========

code	Explanation
`def make_uppercase(func):`	Defines a **decorator function** named make_uppercase. It takes one argument, typically a function, which we call func.
`def wrapper():`	Defines an **inner function** named wrapper inside make_uppercase. This function will hold the modified logic.
`return func().upper()`	Inside wrapper, it **calls the original function** (func()), which returns a string ("hello world"), and then calls the **.upper() method** on that string, making it "HELLO WORLD".
`return wrapper`	The make_uppercase function **returns the inner wrapper function**. It does *not* execute wrapper, it just passes the function object back.
	(Empty line for spacing)
`@make_uppercase`	This is the **decorator syntax**. It is equivalent to saying say_hello = make_uppercase(say_hello).
`def say_hello():`	Defines the original function named say_hello. Because of the decorator on line 6, the **name say_hello is immediately reassigned** to the wrapper function returned by make_uppercase (from line 4).
`return "hello world"`	This is the body of the original function. When the function is called on line 10, this is the string that func() (from line 3) will return.

	(Empty line for spacing)
print(say_hello())	Since this is now the wrapper function, th say_hello (line 8) to get "hello world", co tion then displays this result.

Real-Life Software Use Cases of the Decorator Pattern

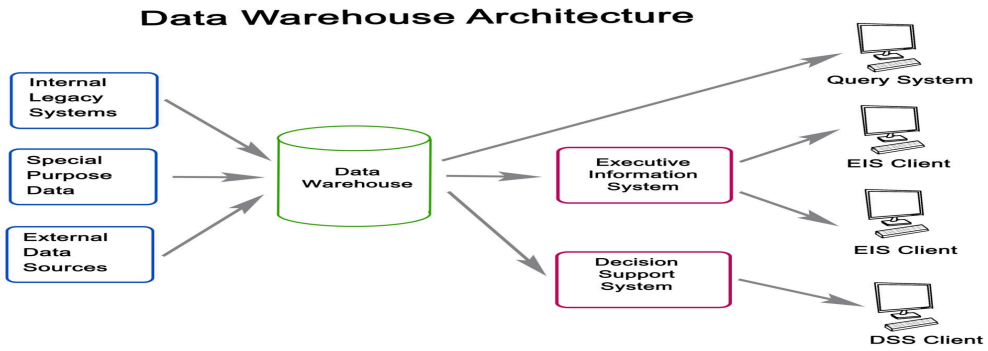
1. Authentication and Authorization (Web Frameworks)

This is perhaps the most common and vital use case in web development frameworks like Django, Flask, and Pyramid.

Decorators Are Used

Web applications need to check if a user is logged in (authentication) and if that user has the necessary permissions to access a resource (authorization) before running the function that handles the request.

2. Caching and Memoization : Decorators are excellent for performance optimization by implementing caching



3. Logging and Metrics

Adding detailed logging or sending metrics (like execution time or error counts) to monitoring systems (like Prometheus or DataDog).

4 Retries and Error Handling

(F) Facade Pattern
==========
===For Example======

```
class CPU:
    def start(self): print("CPU started")

class Memory:
    def load(self): print("Memory loaded")

class ComputerFacade:
    def start_computer(self):
        CPU().start()
        Memory().load()

pc = ComputerFacade()
pc.start_computer()
```

=======Explain The Code==========

Code	Explanation
class CPU:	Defines a class named `CPU`, representing one subsystem component.
def start(self):	Defines a method within the `CPU` class to simulate the CPU starting up.
print("CPU started")	The action performed by the `start` method.
	(Empty line for spacing)
`class Memory:`	Defines another class named `Memory`, representing another subsystem component.
def load(self):	Defines a method within the `Memory` class to simulate data loading.
print("Memory loaded")	The action performed by the `load` method.**8**(Empty line for spacing)
	(Empty line for spacing)
`class ComputerFacade:`	Defines the central **Facade** class. This class provides a simplified interface to the complex subsystem.
def start_computer(self):	Defines the main, simple method that the client (user) will call.

`CPU().start()`	Inside the facade method, it **instantiates** the `CPU` class and immediately calls its complex method (`start()`).
`Memory().load()`	It **instantiates** the `Memory` class and immediately calls its complex method (`load()`).
	(Empty line for spacing)
pc = ComputerFacade()	**Client Code:** Creates an instance of the `ComputerFacade` class and assigns it to the variable `pc`.
pc.start_computer()	**Client Code:** Calls the simplified `start_computer()` method on the facade object. This triggers the execution of lines 11 and 12.

Real-Life Software Use Cases of the Decorator Pattern

1. API and Library Simplification
2. System Initialization and Configuration
3. Transaction Management and Workflows
4. Layering and Decoupling

(G) Behavioral Design Patterns

These patterns deal with communication between objects.

==========

==========For Example==========

```
class Observer:
    def update(self, msg):
        print("Received:", msg)

class Subject:
    def __init__(self):
        self.observers = []

    def add(self, obs):
```

```
        self.observers.append(obs)

    def notify(self, msg):
        for obs in self.observers:
            obs.update(msg)

s = Subject()
s.add(Observer())
s.notify("Data Updated")
```

Code	Explanation
class Observer:	Defines the **Observer** class. Observers are the components that are interested in changes and will receive updates.
def update(self, msg):	Defines the required method for an observer. When the Subject has a change, it calls this method on all registered observers, passing the update message (`msg`).
`print("Received:", msg)`	The action taken by this specific observer: it prints the message it received.
class Subject:	Defines the **Subject** (or Observable) class. The Subject is the component that holds the data of interest and notifies others when that data changes.
def __init__(self):	The constructor for the `Subject` class.
`self.observers = []`	Initializes an **empty list** named `observers`. This list will store all the `Observer` objects that are currently subscribed to this Subject.
def add(self, obs):	Defines a method to **register (subscribe)** a new Observer object.
self.observers.append(obs)	Appends the passed-in Observer instance (`obs`) to the internal list of subscribers.
def notify(self, msg):	Defines the method that the Subject calls when its state changes, triggering a **notification** to all subscribed observers.
for obs in self.observers:	Iterates through every Observer object currently stored in the list.

obs.update(msg)	For each subscribed Observer, it **calls its `update()` method**, passing the notification message (`msg`).
`s = Subject()`	**Client Code:** Creates an instance of the `Subject` class and assigns it to the variable `s`.
`s.add(Observer())`	**Client Code:** Creates a new instance of the `Observer` class and registers it with the `Subject` instance `s` by calling the `add()` method.
s.notify("Data Updated")	**Client Code:** Calls the `notify()` method on the Subject instance `s`, passing the string `"Data Updated"`. This triggers the loop on line 13, which calls the `update()` method of the registered Observer.

Real-Life Software Use Cases of the Behavioral Design Patterns

1. Chain of Responsibility

This pattern creates a chain of receiver objects for a request. Each receiver decides either to process the request or pass it to the next receiver in the chain.

2. Command

This pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

3. Iterator

This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation

4. Observer (or Publish-Subscribe)

This pattern defines a one-to-many dependency so that when one object (the Subject) changes state, all its dependents (the Observers) are notified and updated automatically.

5. Strategy

This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

(H) Strategy Pattern :

Define multiple algorithms and let the user choose."
==========

```
class Add:
    def execute(self, a, b): return a + b

class Multiply:
    def execute(self, a, b): return a * b

class Context:
    def __init__(self, strategy):
        self.strategy = strategy

    def perform(self, a, b):
        return self.strategy.execute(a, b)

ctx = Context(Add())
print(ctx.perform(5, 3))

ctx = Context(Multiply())
print(ctx.perform(5, 3))
```

==========Explain the code==========

class Add:	Defines the first **Concrete Strategy** class. This class implements one specific algorithm (addition).
`def execute(self, a, b):`	Defines the required method (`execute`) that all strategy classes must implement. This is the **Strategy Interface** method.
return a + b	The implementation of the addition algorithm.
`class Multiply:`	Defines the second **Concrete Strategy** class. This implements another algorithm (multiplication).
def execute(self, a, b):	Implements the same required method (`execute`).

return a + b	The implementation of the addition algorithm.
class Multiply:	Defines the second **Concrete Strategy** class. This implements another algorithm (multiplication).
def execute(self, a, b):	Implements the same required method (`execute`).
return a * b	The implementation of the multiplication algorithm.
(Empty line for spacing)**9**`class Context:`	Defines the **Context** class. This class holds a reference to a Strategy object and interacts with it.
def __init__(self, strategy):	The constructor takes one argument, an instance of a Strategy class (either `Add` or `Multiply`).
self.strategy = strategy	Stores the passed-in Strategy object internally. This allows the Context to swap algorithms at runtime.
def perform(self, a, b):	Defines the main method the client calls to execute an operation.
`return self.strategy.execute(a, b)`	The Context **delegates** the work to the stored Strategy object by calling its standard `execute()` method. The Context doesn't know *how* the calculation is performed, only that it can call `execute()`.
ctx = Context(Add())	**Client Code:** Creates a new `Context` instance. It passes a new `Add` object as the initial strategy. `ctx` is now configured for addition.
`print(ctx.perform(5, 3))`	Calls the perform method on the Context. The Context delegates to the Add strategy: $5 + 3 = 8
ctx = Context(Multiply())	**Client Code:** Creates a *new* `Context` instance (or reassigns the strategy). It passes a new `Multiply` object. `ctx` is now configured for multiplication.
print(ctx.perform(5, 3))	Calls the perform method again. The Context now delegates to the Multiply strategy: $5 \times 3 = 15

Real-Life Software Use Cases of the Strategy Pattern

1. Payment Processing Systems
2. File Compression and Archiving
3. Data Validation and Formatting

4. Rendering and Layout Algorithms
5. Sorting and Searching

(i) Command Pattern :

Encapsulate a request as an object.
==========

For Example :

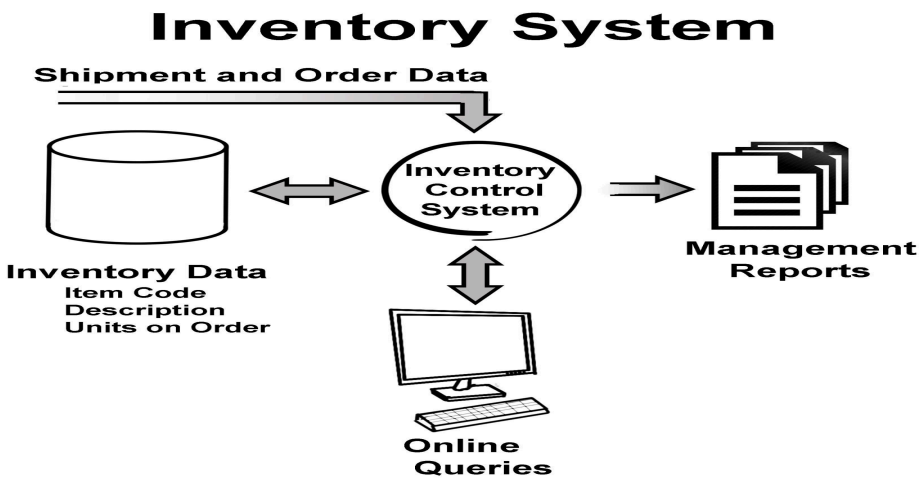
```
class Light:
    def on(self): print("Light ON")
    def off(self): print("Light OFF")

class Command:
    def execute(self): pass

class TurnOn(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.off()

light = Light()
cmd = TurnOn(light)
cmd.execute()
```

===Explanation the Code =======

Code	Explanation
def on(self): print("Light ON")	Defines the action to turn the light **on**.
def off(self): print("Light OFF")	Defines the action to turn the light **off**.
`def execute(self): pass`:	Declares an abstract `execute` method that all concrete commands must implement. This is the method that invokes the Receiver's action.
class TurnOn(Command):	It inherits from the `Command` base class.
`def __init__(self, light):`	The constructor takes a **Receiver object** (a

`self.light = light`:	`Light` instance) and stores it internally.
`def execute(self):` `self.light.off()`:	**This is the point of confusion/inconsistency.** Since the class is named `TurnOn`, its `execute` method is expected to call `self.light.on()`. However, the code as written calls `self.light.off()`.
`light = Light()`:	An instance of the **Receiver** object is created.
`cmd = TurnOn(light)`:	An instance of the **Concrete Command** is created, taking the `light` object as its receiver.
`cmd.execute()`:	The `execute` method of the `cmd` object is called.

The Command Design Pattern (Conceptual):



The Command Pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Client (The main script): Creates a Concrete Command and sets its Receiver.

Invoker (Not present in this simple script, but usually a button/remote): Holds a reference to the Command and tells it to execute.

Command (The `Command` class): Declares the interface for executing an operation.

Concrete Command (`TurnOn` class): Implements the `execute` method by calling the action on the Receiver.

Receiver (`Light` class): Knows how to perform the operations.

Real-Life Software Use Cases of the Command Pattern

1. Undo/Redo Functionality
2. Macro Recording and Execution
3. Asynchronous Task Queues and Scheduling
4. Parameterizing UI Elements and Buttons
5. Remote Procedure Calls (RPC) and Networking